# 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.

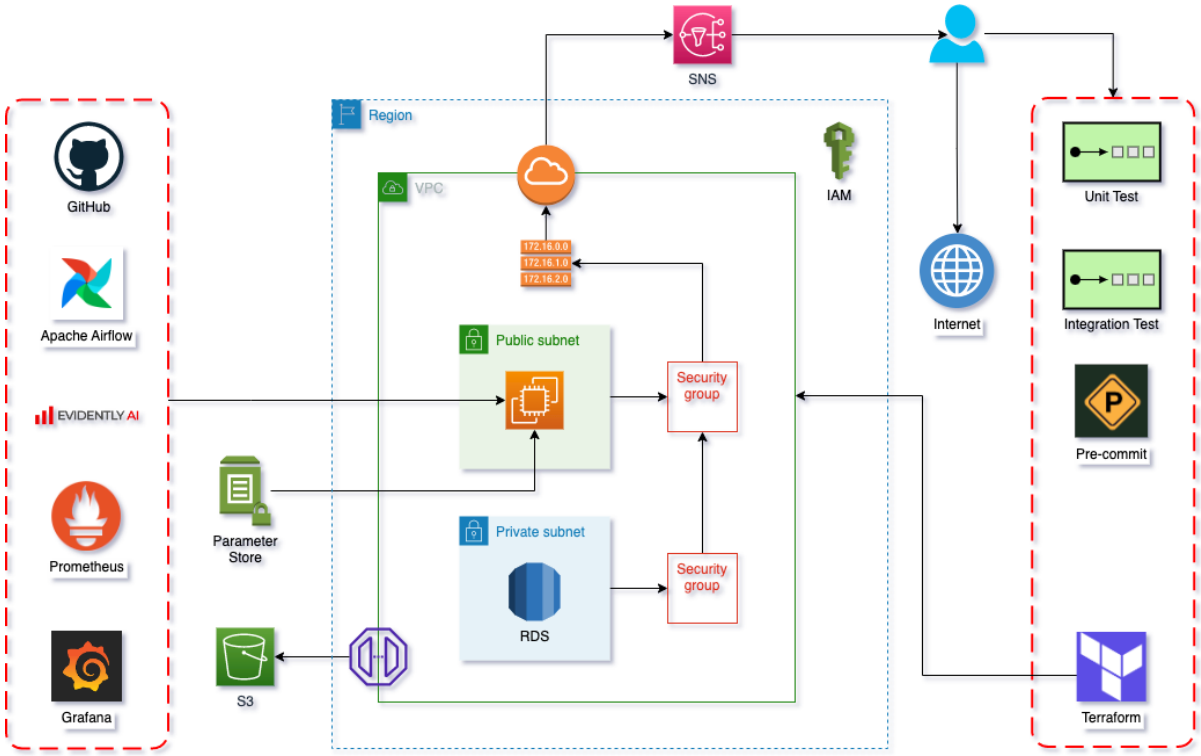



**Exhibit-1: MLOps Project Diagram (Image by Author)**

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.

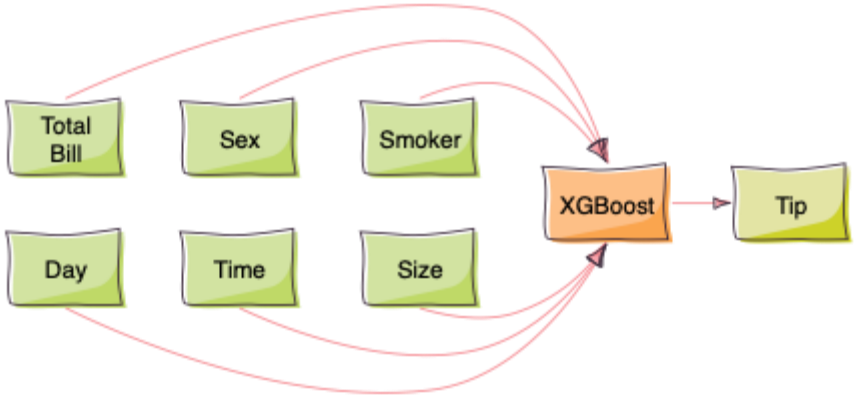



Exhibit-3: Model, Features, and Label (Image by Author)

**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 rawdatafrom 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 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:

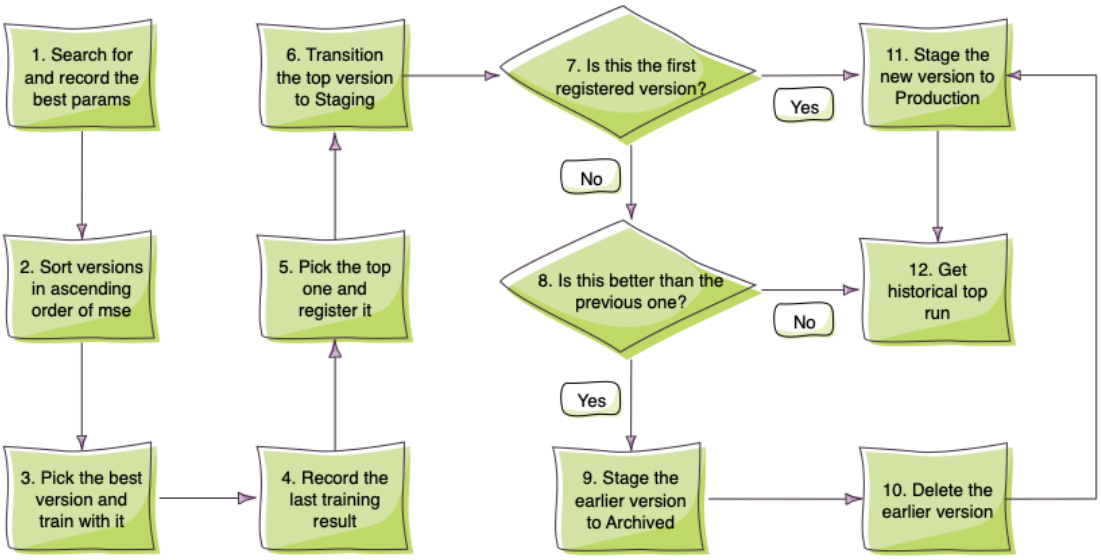



Exhibit-5: Flow of Logic of MLflow Tasks (Image by Author)

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 usescope.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. Thehp.loguniform gives learning rate and L2 regularization parameters as multiples of power, similar to, 0.1, 0.01, or 0.001.
Thehp.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.

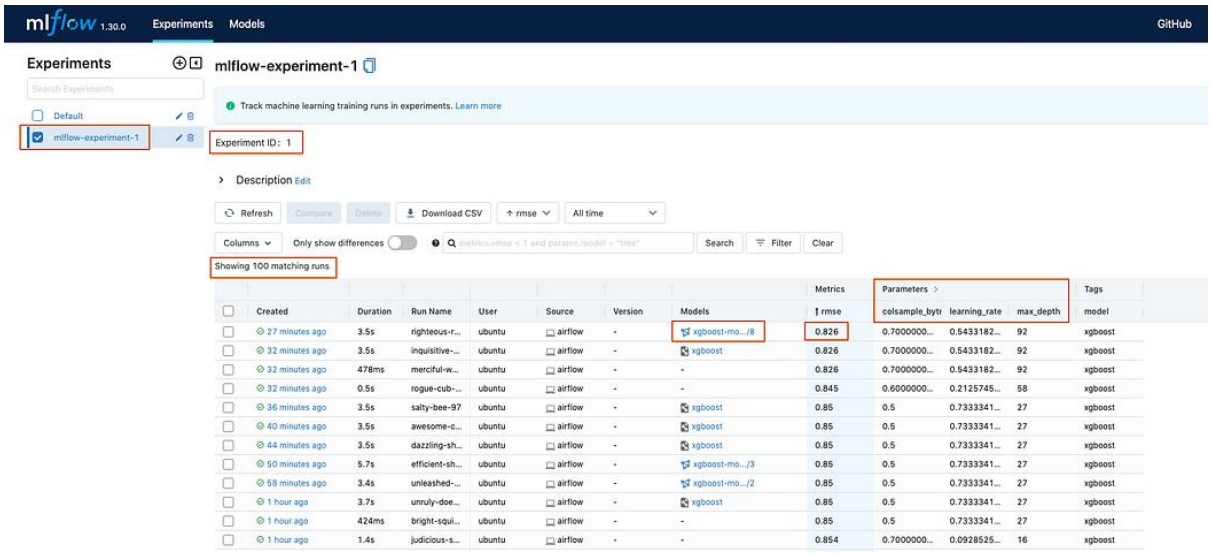



Exhibit-6: Records of MLflow Runs (Image by Author)

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 thesearch_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:

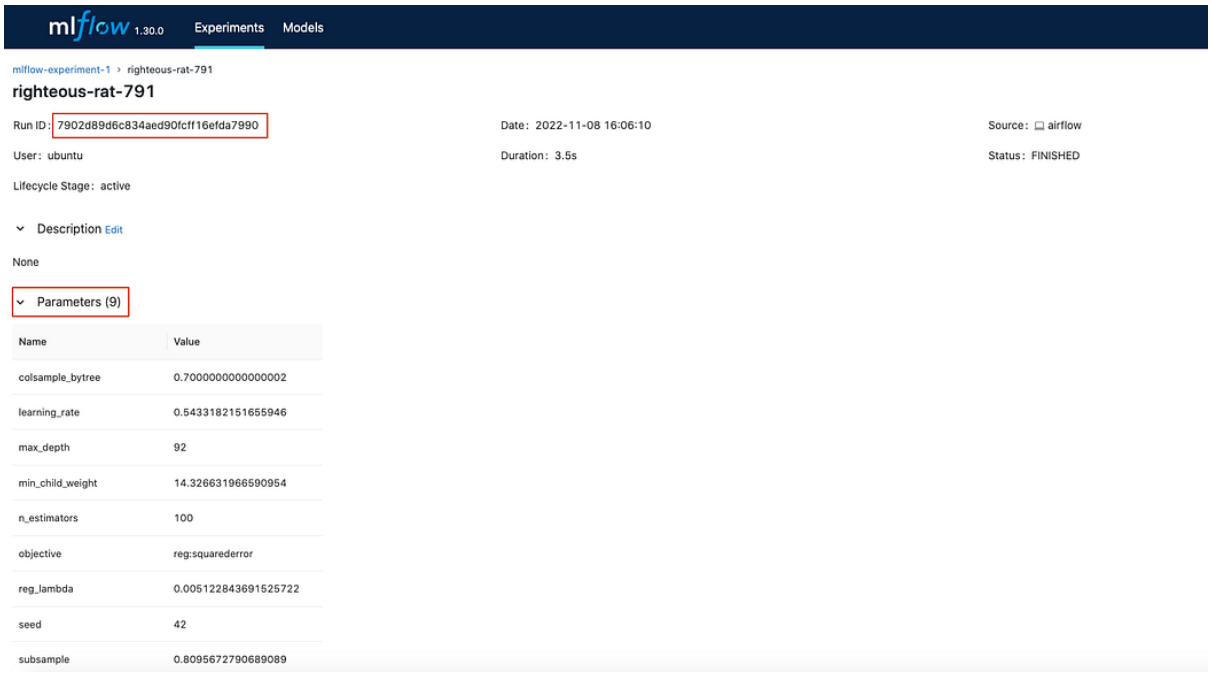



Exhibit-7: Details of a Run (Image by Author)

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.

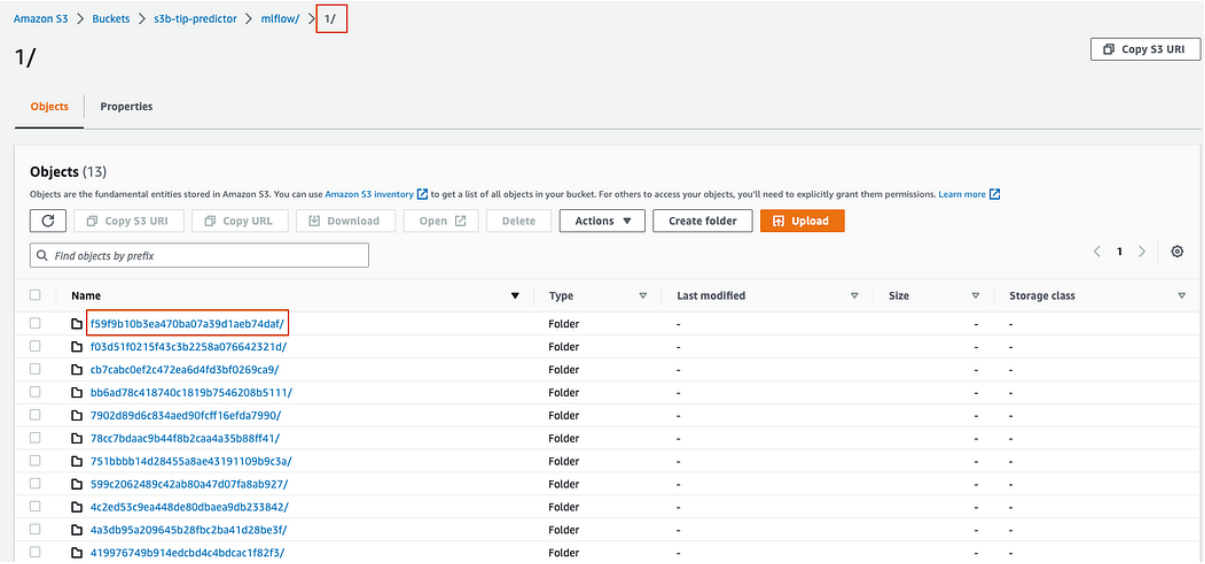



Exhibit-8: Runs in the Experiment Folder in S3 (Image by Author)

And each run stores artifacts as explained earlier:

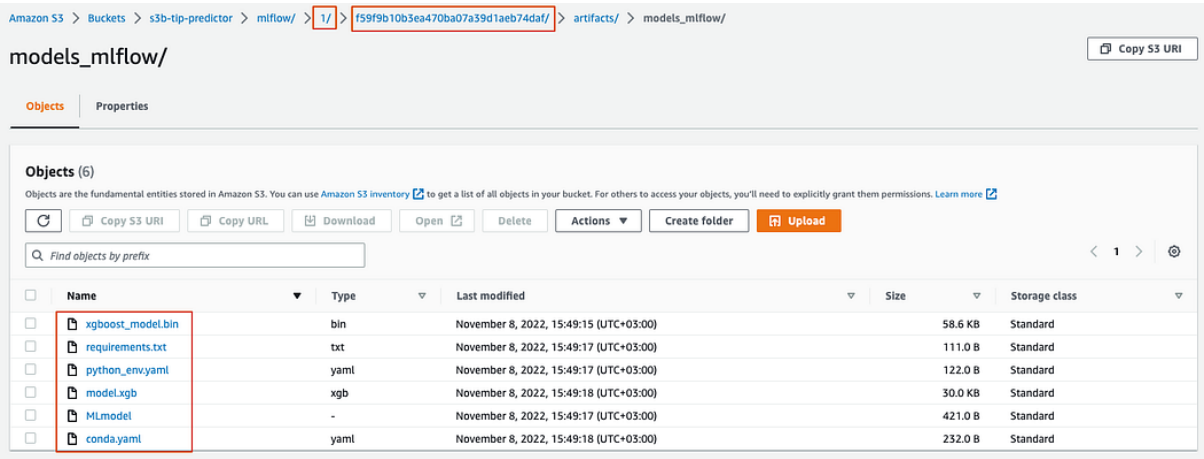



Exhibit-9: Run Folder in S3 (Image by Author)

**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:

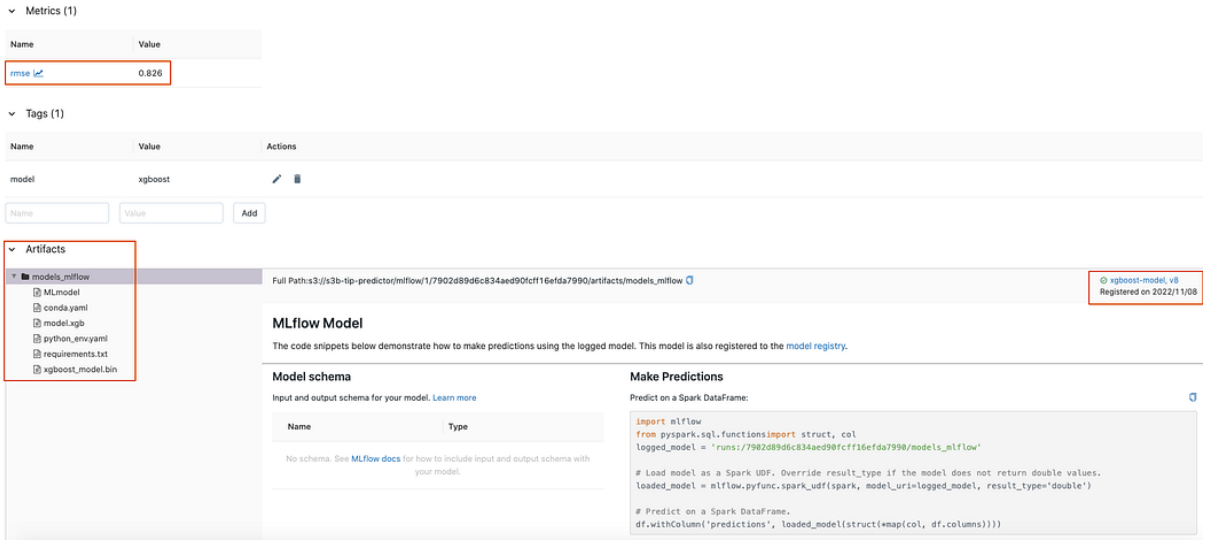



Exhibit-10: Model and Artifacts of a Run on MLflow UI (Image by Author)

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:

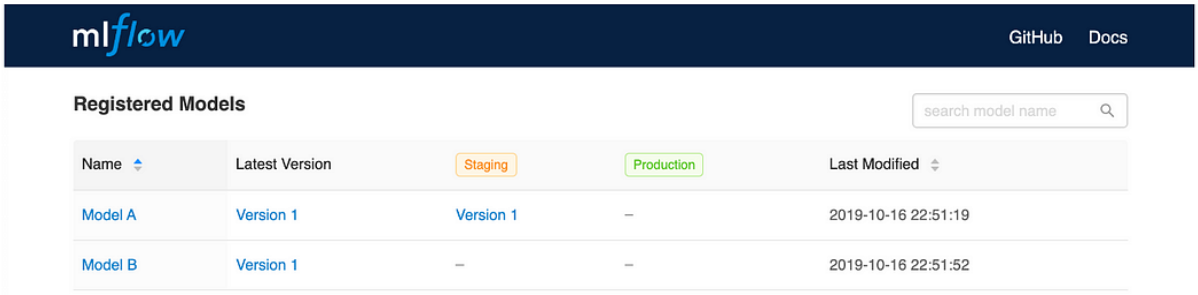



Exhibit-11: MLflow Stage Transition (Taken from https://www.mlflow.org/docs/latest/model-registry.html)

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 theload_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="rmses", 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="rmses", 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 thebest_rmse_dict via email through *AWS SNS*. The notification we'll receive can be seen in Exhibit 12:

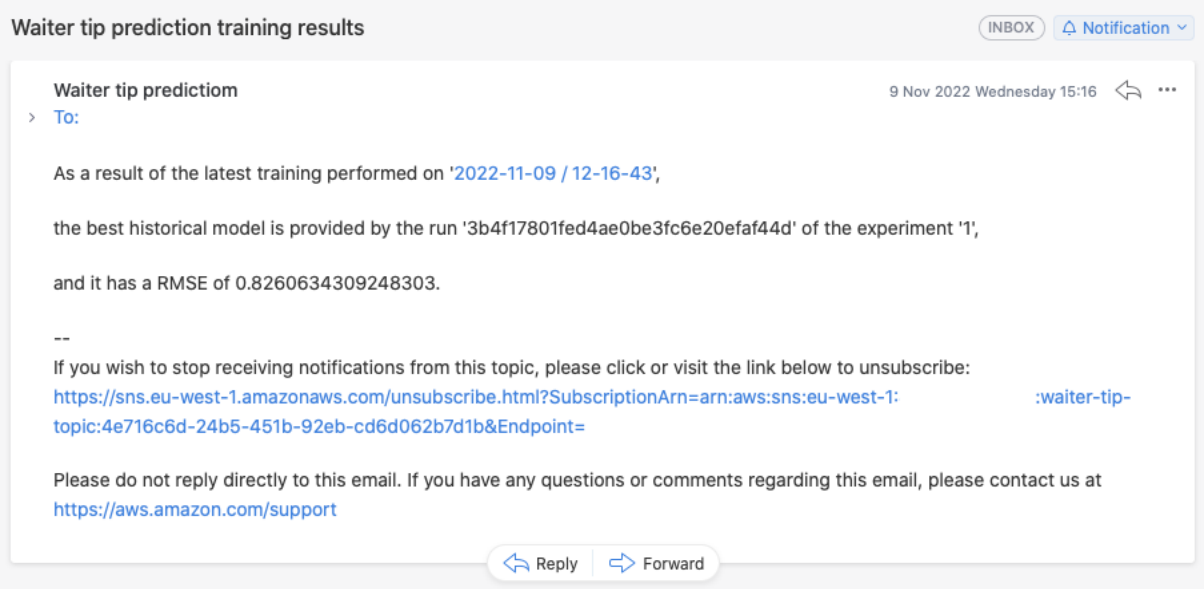



Exhibit-12: Training Results Notified by AWS SNS (Image by Author)

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.**

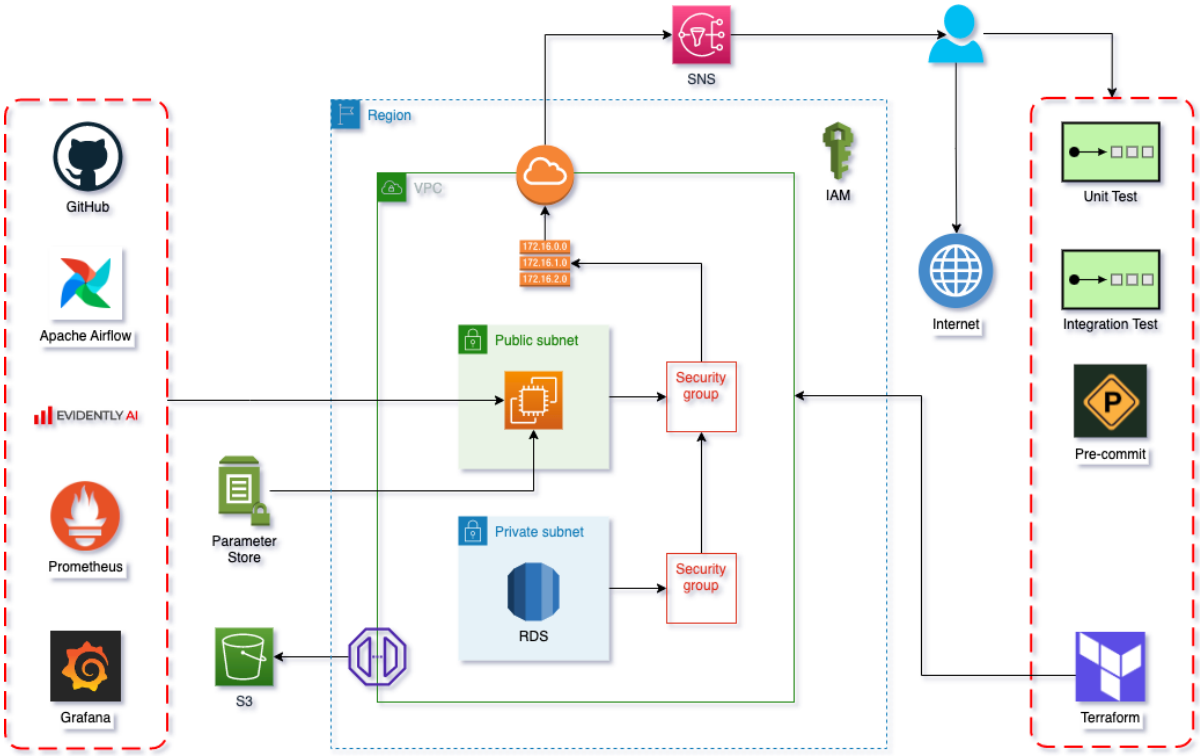



**Exhibit-1: MLOps Project Diagram**

**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:**

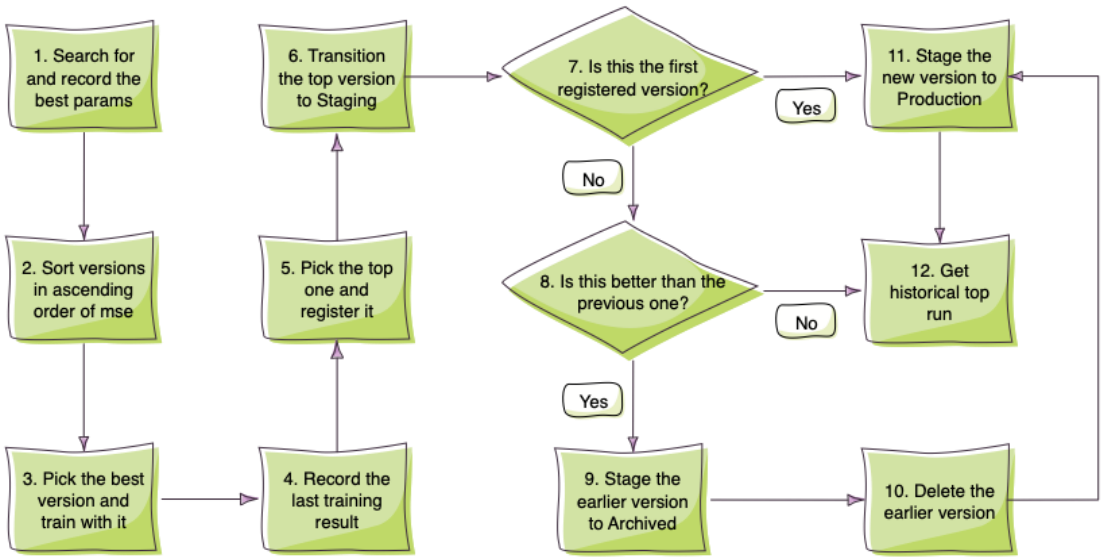



**Exhibit-2: Flow of Logic of MLflow Tasks (Image by Author)**

**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:**

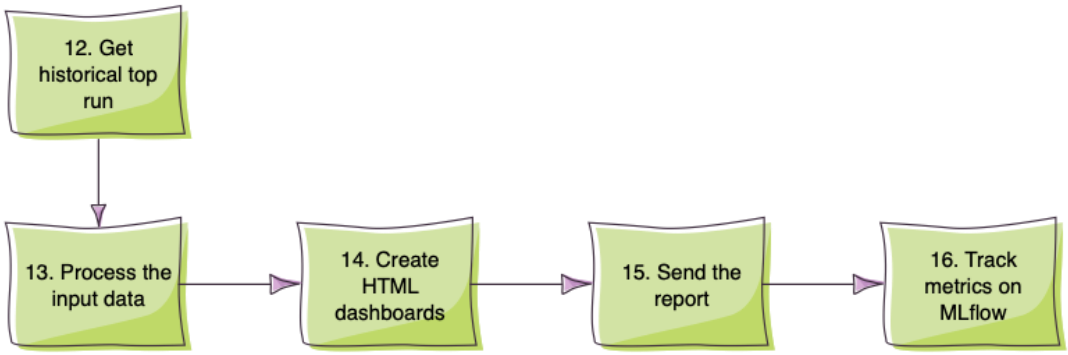



**Exhibit-3: Evidently Part of the Flow of Logic (Image by Author)**

**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].**

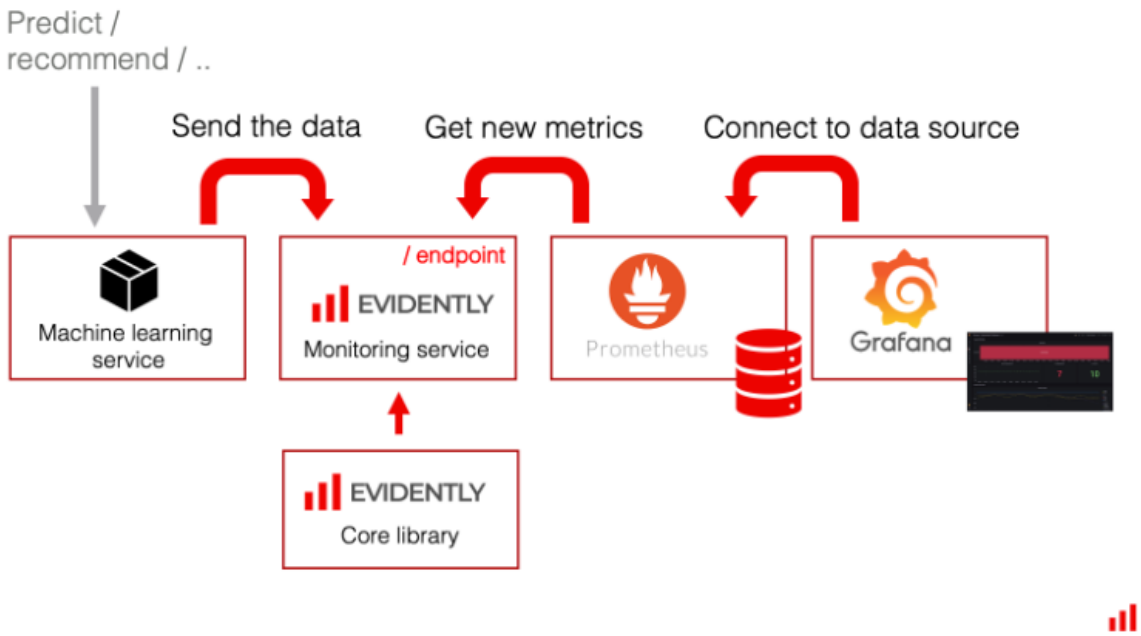



**Exhibit-4: High-level Overview of the Integration Architecture**

***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.

Search

| 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_unique <= 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].***

Data Summary

| | reference | current |
|---|---|---|
| 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.**

day
cat

| | reference | current |
|---|---|---|
| count | 182 | 60 |
| unique | 4 (2.2%) | 4 (6.67%) |
| most common | 2 (36.26%) | 3 (33.33%) |
| missing | 0 (0.0%) | 0 (0.0%) |
| new categories | | 0 |
| missing categories | | 0 |

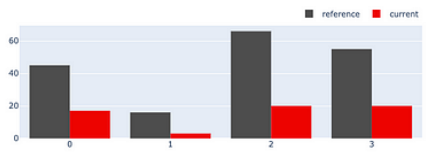



**Exhibit-7: Feature Overview Table of Features Widget by Evidently (Image by Author)**

**“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.**

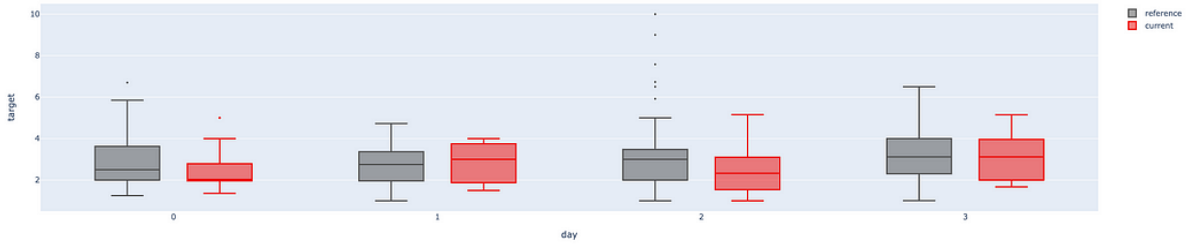



**Exhibit-8: Feature by Target Plot of the Features Widget by Evidently (Image by Author)**

**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) | value ref | value curr | difference | top 5 correlation diff numerical (Spearman) | 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.**

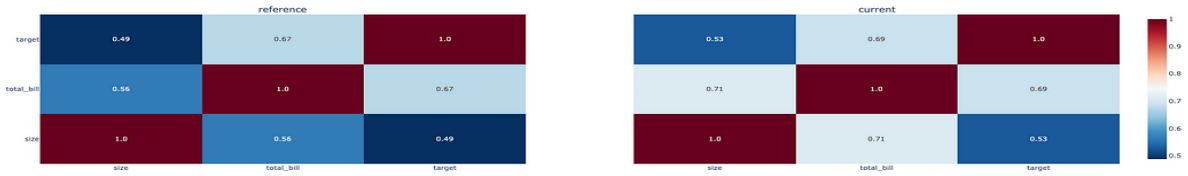



**Exhibit-10: Correlation Heatmaps of the Correlation Widget by Evidently (Image by Author)**

- **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.**

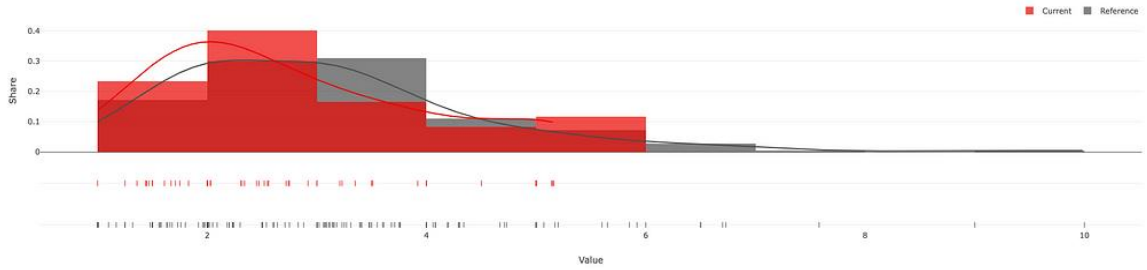



**Exhibit-11: The Target Drift by Evidently (Image by Author)**

**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.**

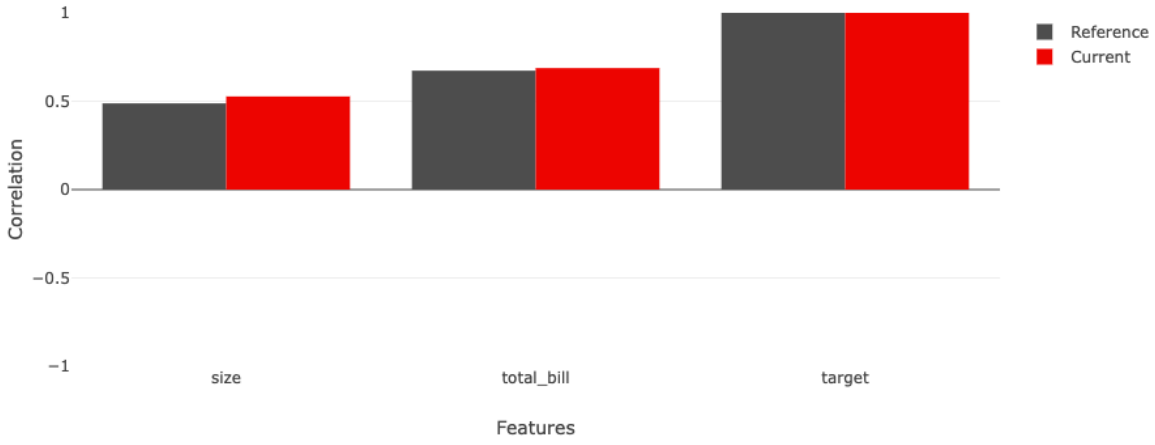



**Exhibit-12: The Target Correlations by Evidently (Image by Author)**

**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.**

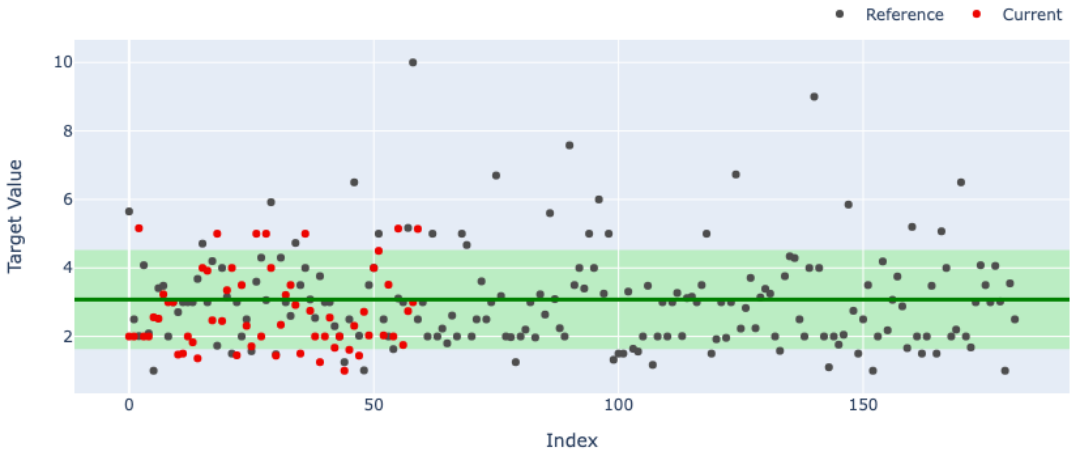



**Exhibit-13: The Target Values by Evidently (Image by Author)**

***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.**

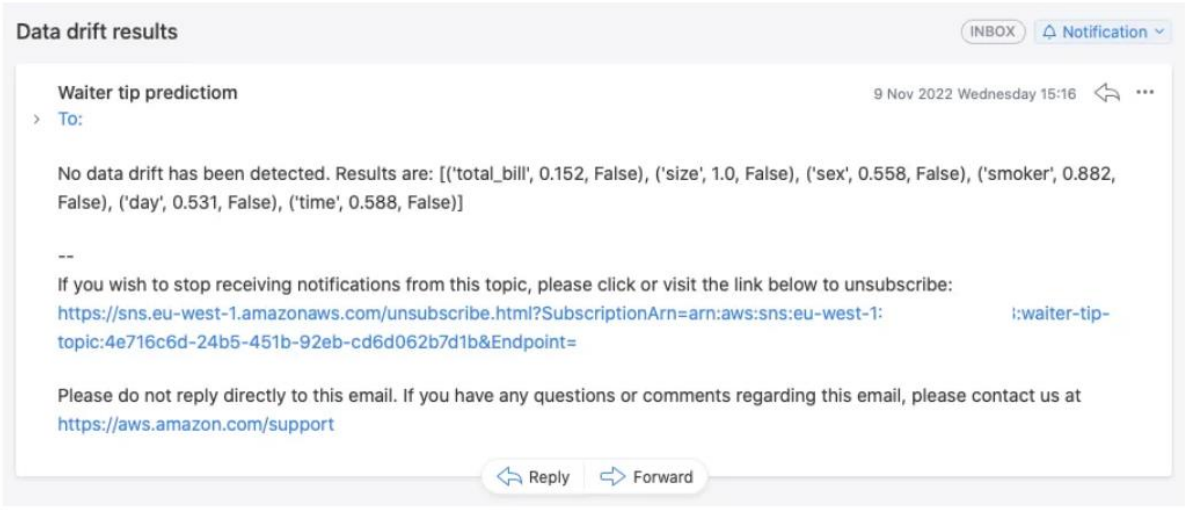



Exhibit-14: Data Drift Summary Results Notified by AWS SNS (Image by Author)

**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, 0.041666666666666664, 0.041666666666666664], [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", "stattest_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, 0.009238728750923877, 0.0, 0.004619364375461938, 0.009238728750923877, 0.004619364375461938], [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", "stattest_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, 3.496000000000004, 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", "stattest_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", "stattest_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", "stattest_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", "stattest_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", "stattest_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", "stattest_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 theDB_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.

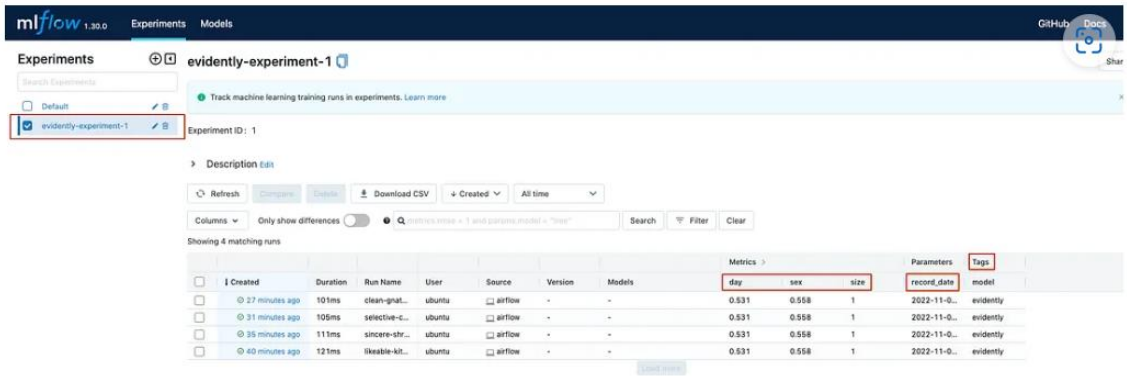



Exhibit-15: Evidently Data Drift Records on MLflow (Image by Author)

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.

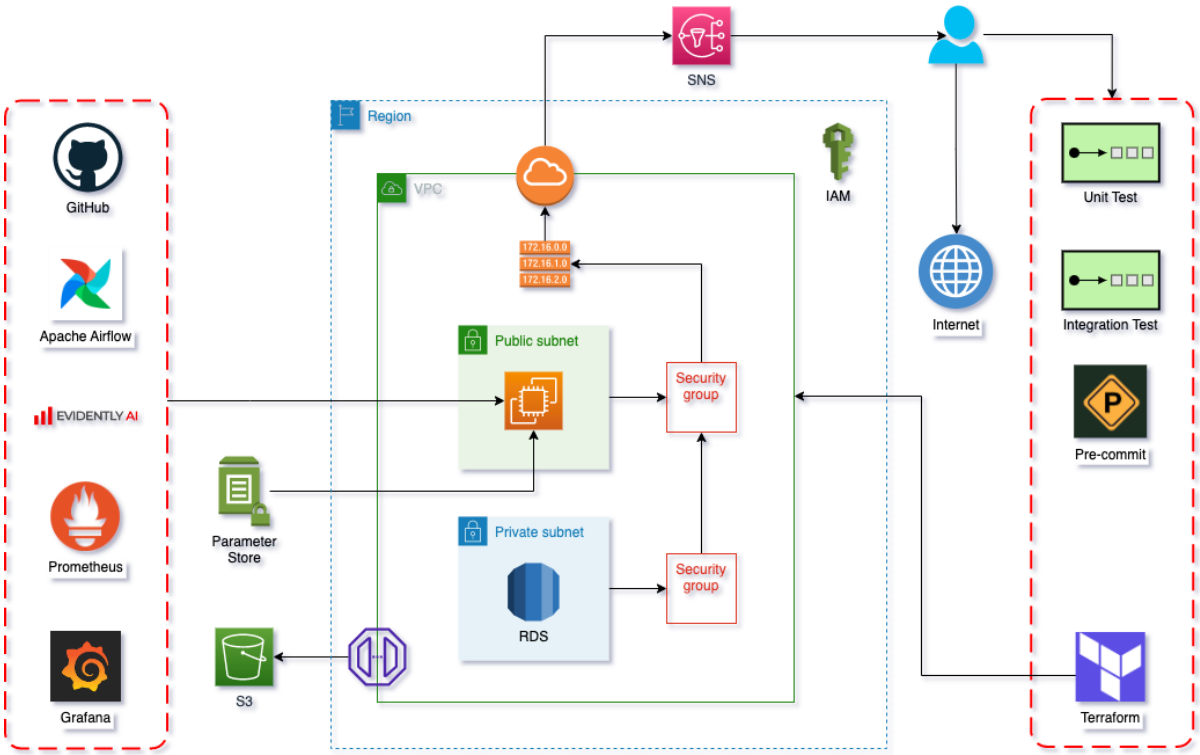



**Exhibit-1: MLOps Project Diagram**

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

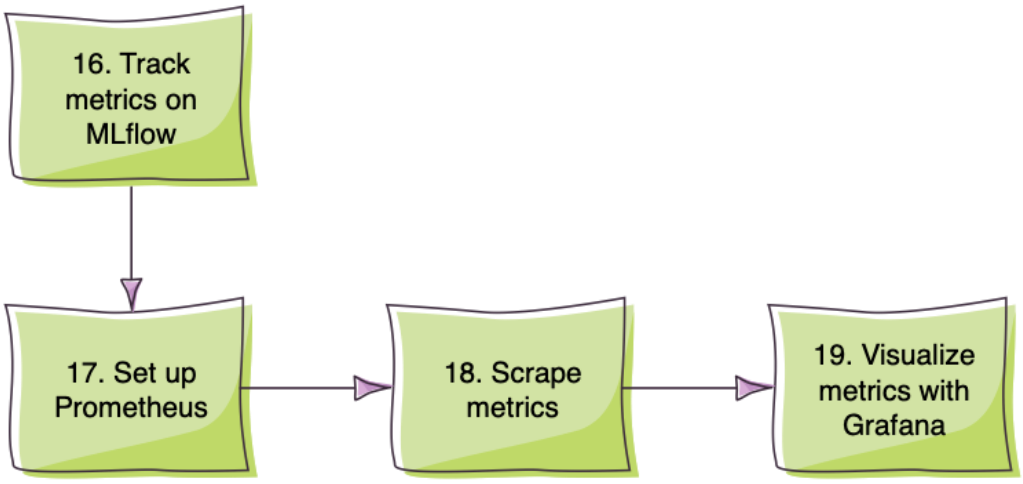



**Exhibit-2: Prometheus and Grafana Parts of the Flow of Logic**

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*.

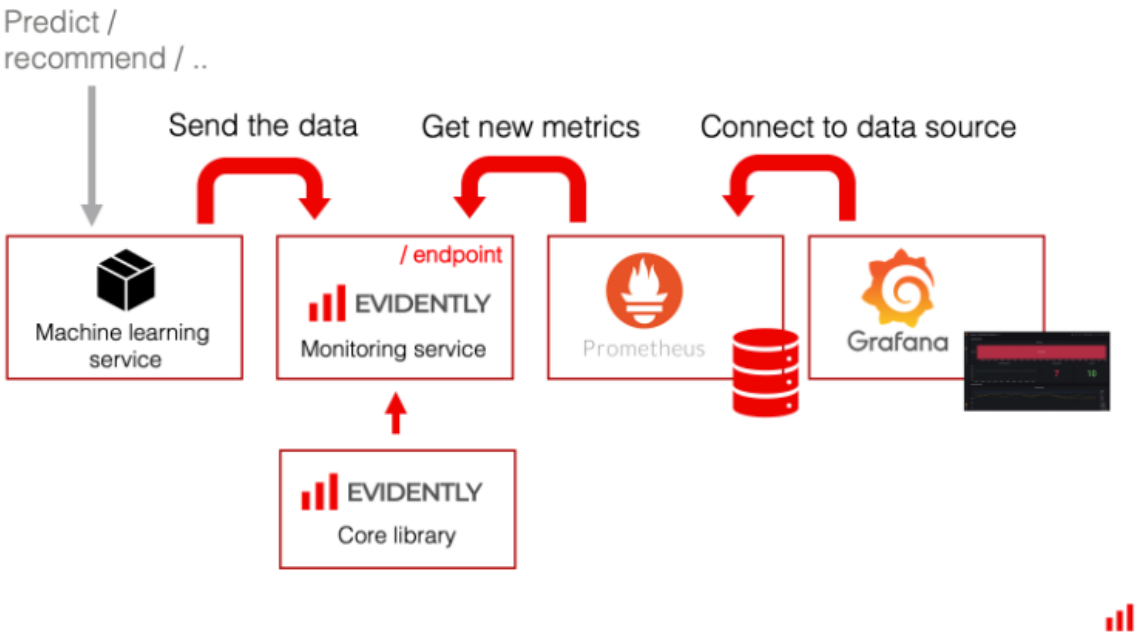



**Exhibit-3: High-level Overview of the Integration Architecture**

**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.

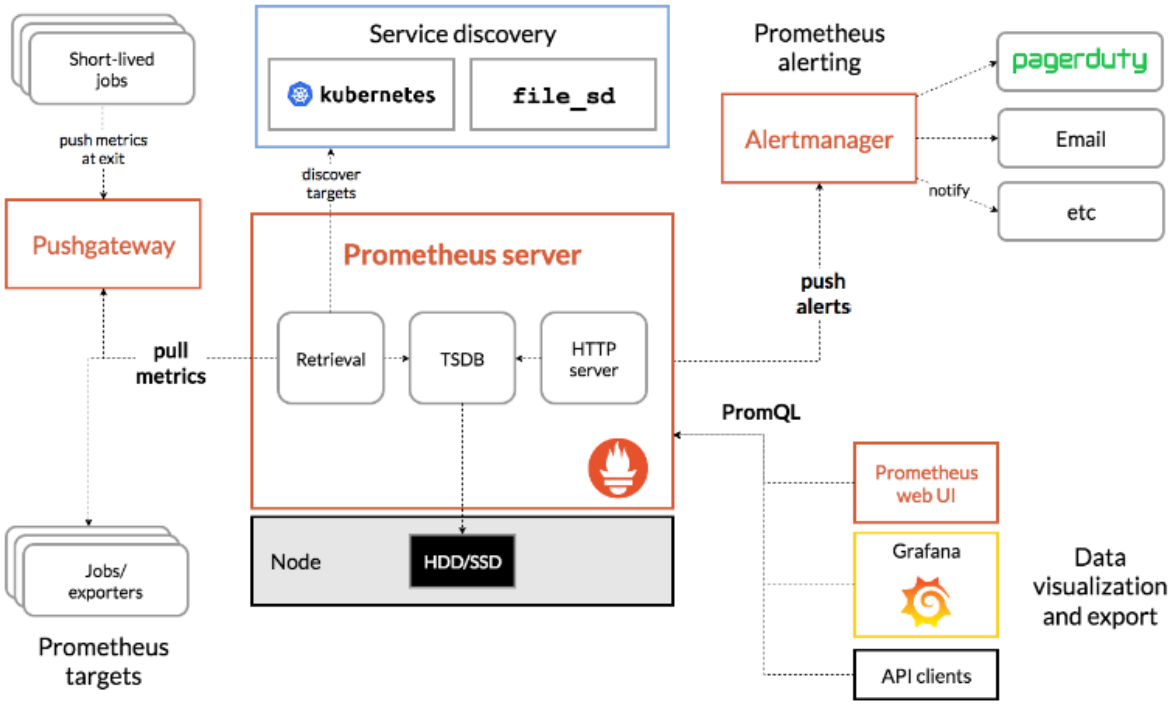



**Exhibit 4: The Architecture of Prometheus and Some of its Ecosystem Components**

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.

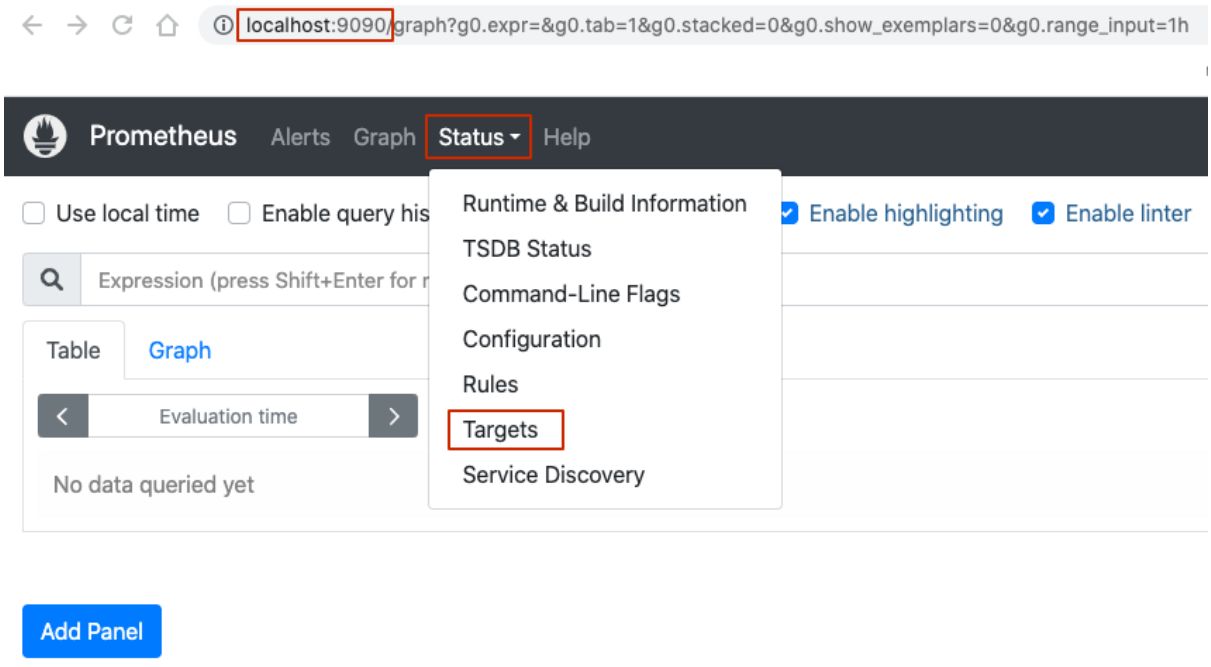



Exhibit-5: Prometheus Web UI (Image by Author)

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

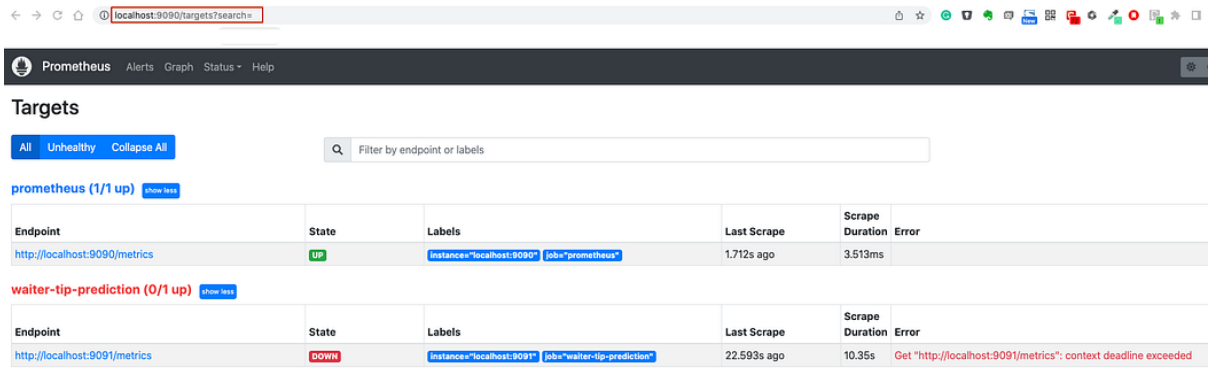



Exhibit-6: Prometheus Targets (Image by Author)

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.

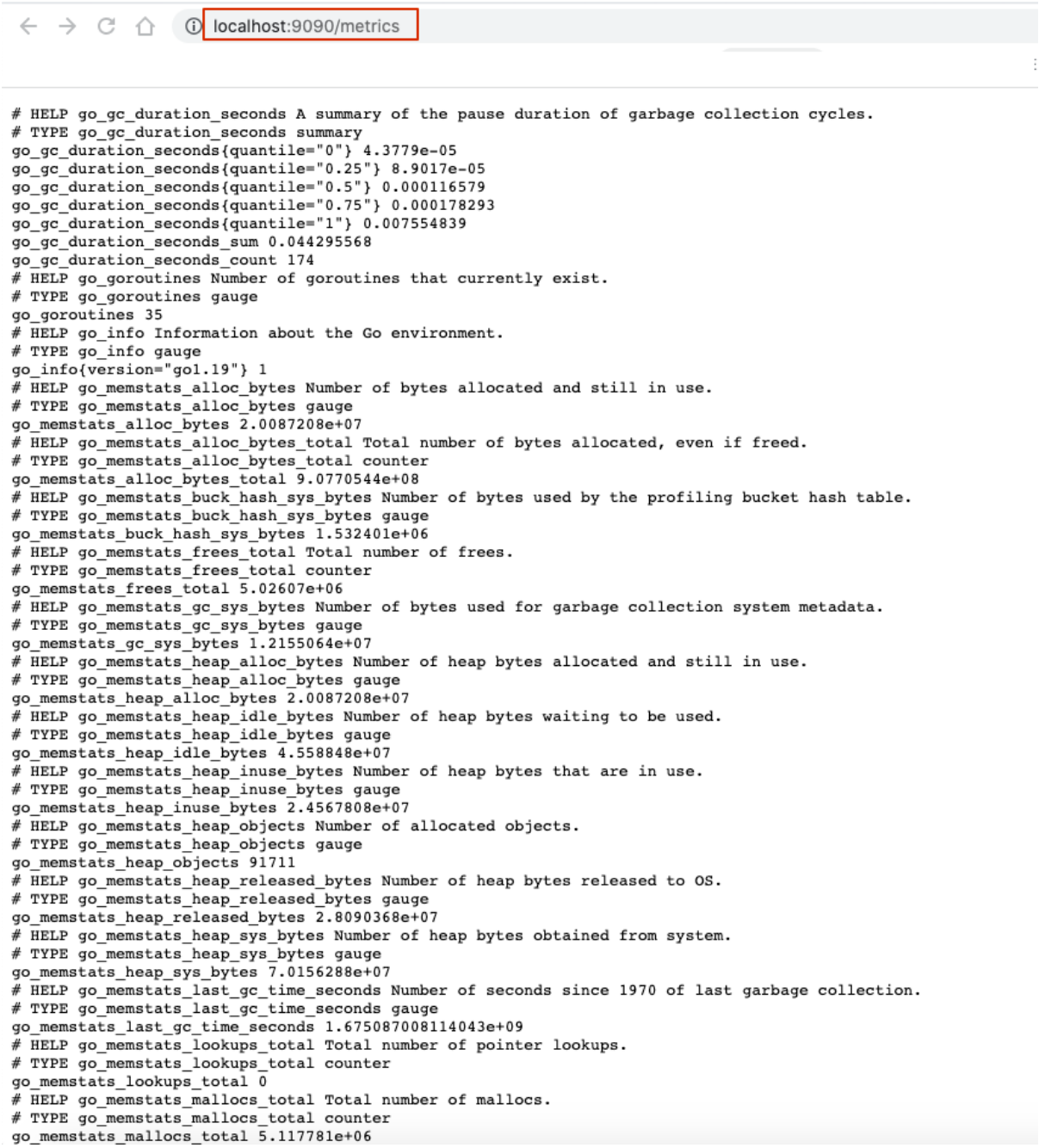



Exhibit-7: Prometheus Metrics (Image by Author)

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].

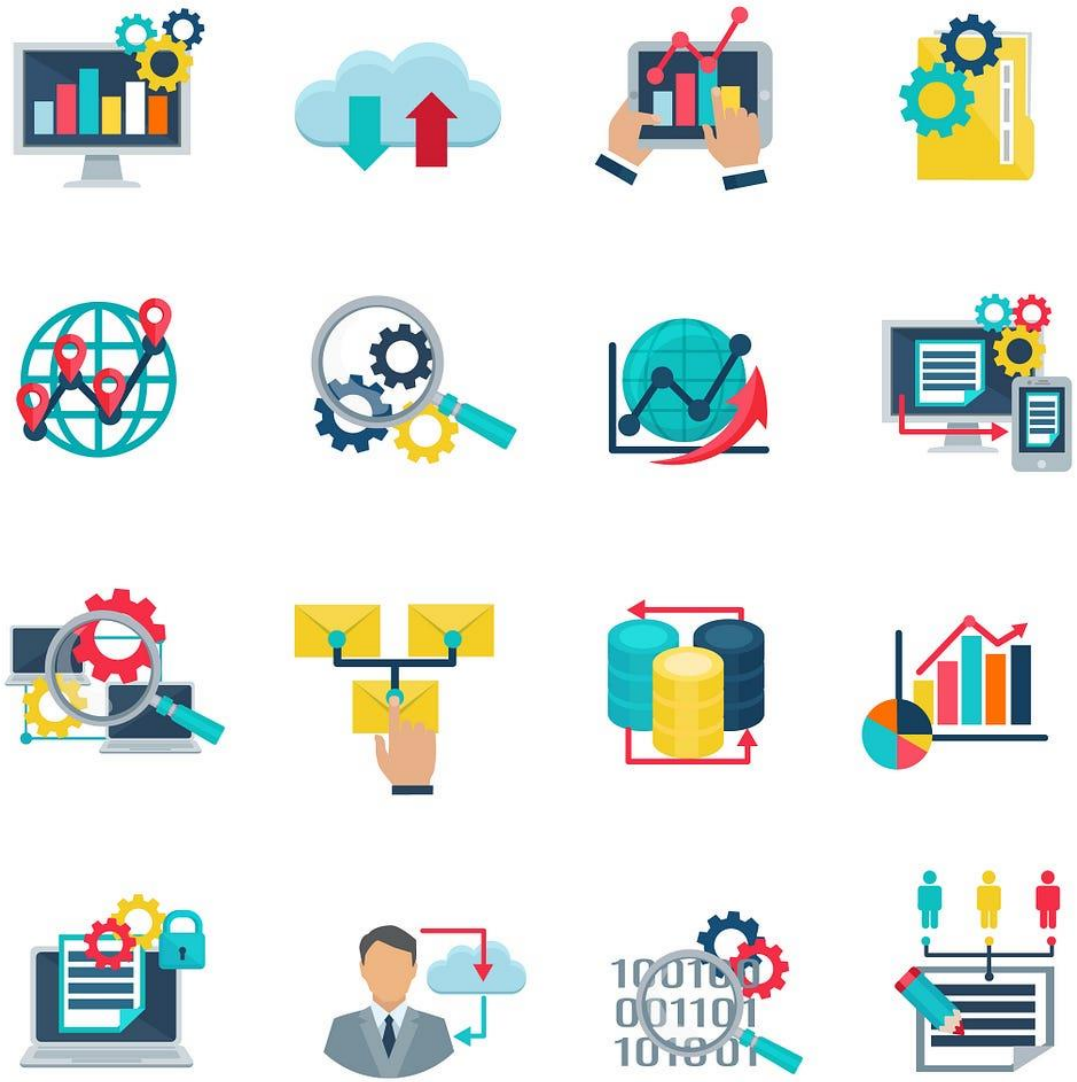


Image by macrovector on Freepik

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:

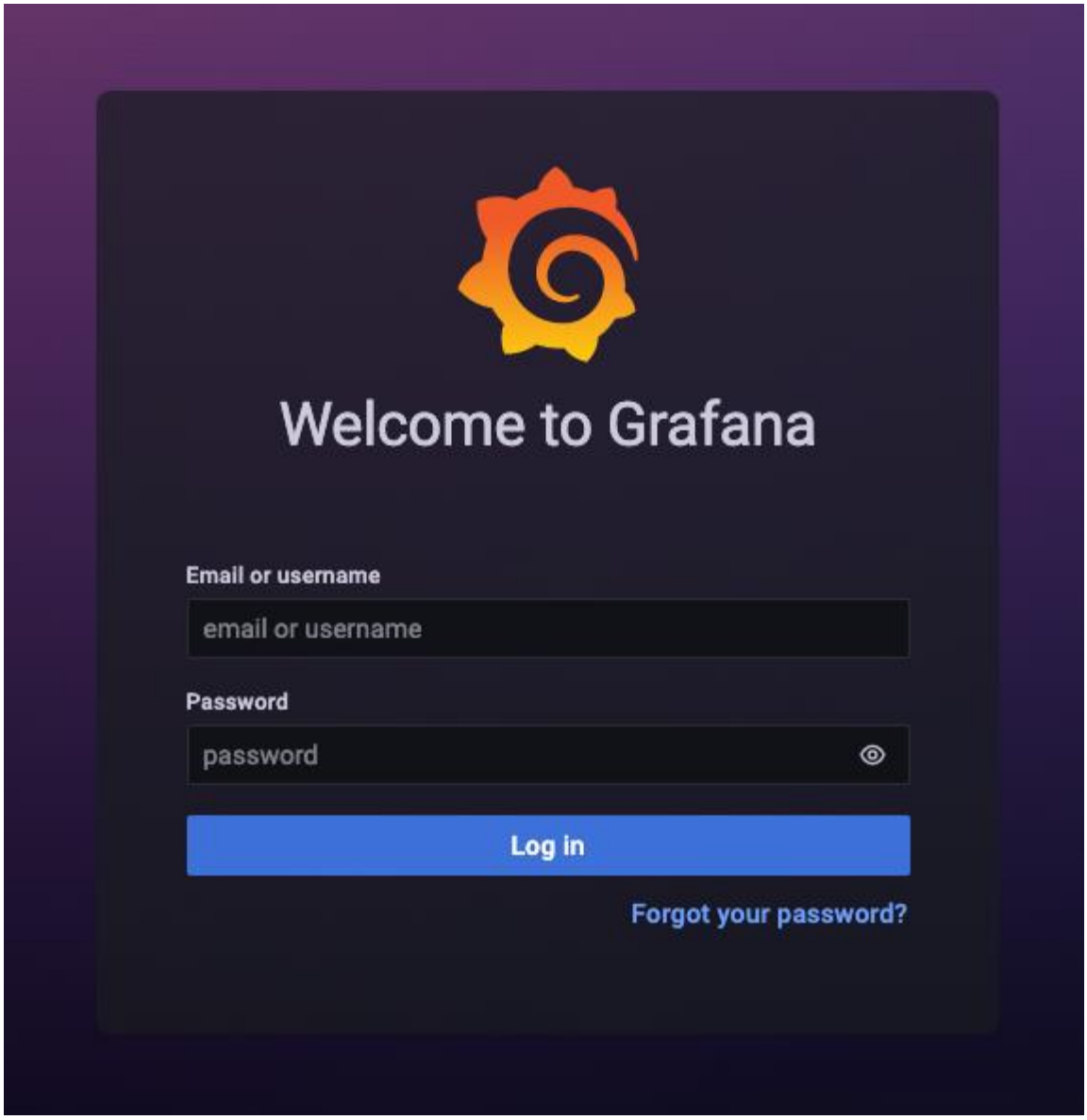



Exhibit-8: Grafana Login (Image by Author)

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.”

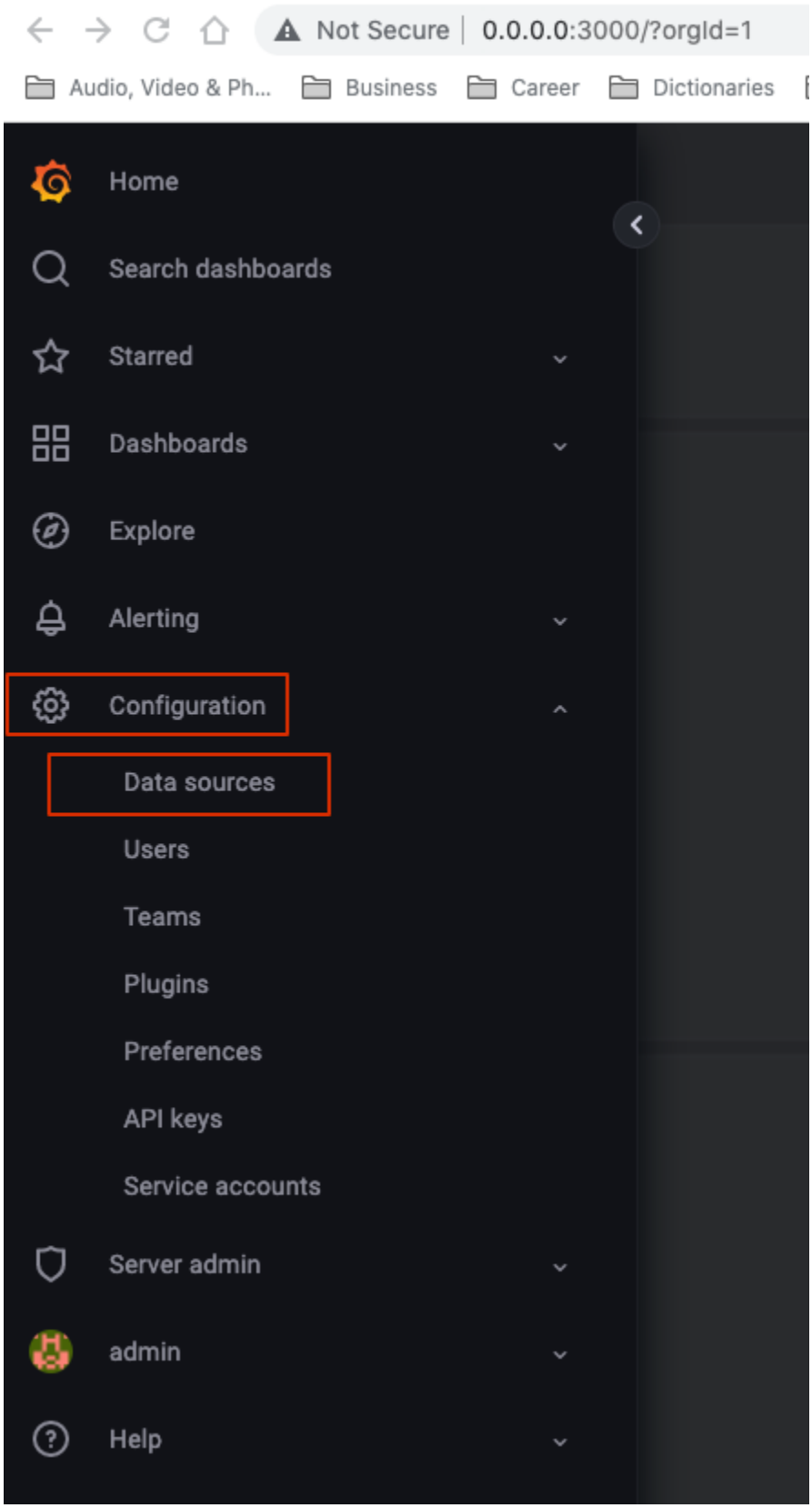



Exhibit-9: Grafana Data Sources (Image by Author)

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

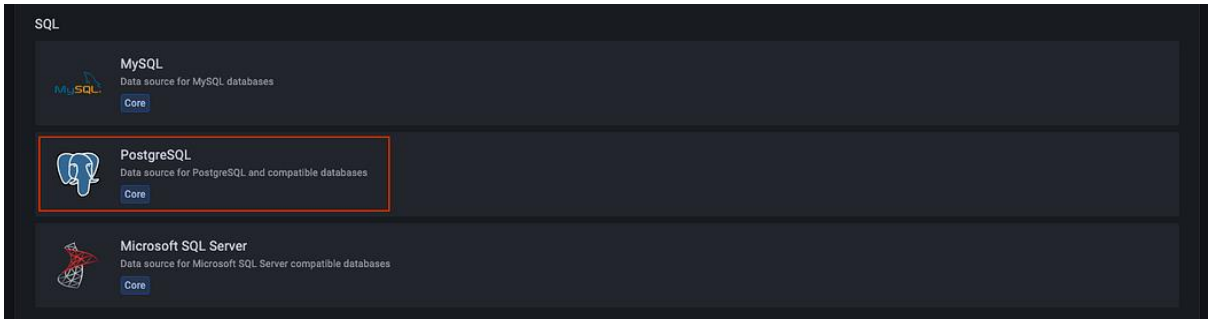



Exhibit-10: PostgreSQL as the Data Source (Image by Author)

Enter the endpoint information:

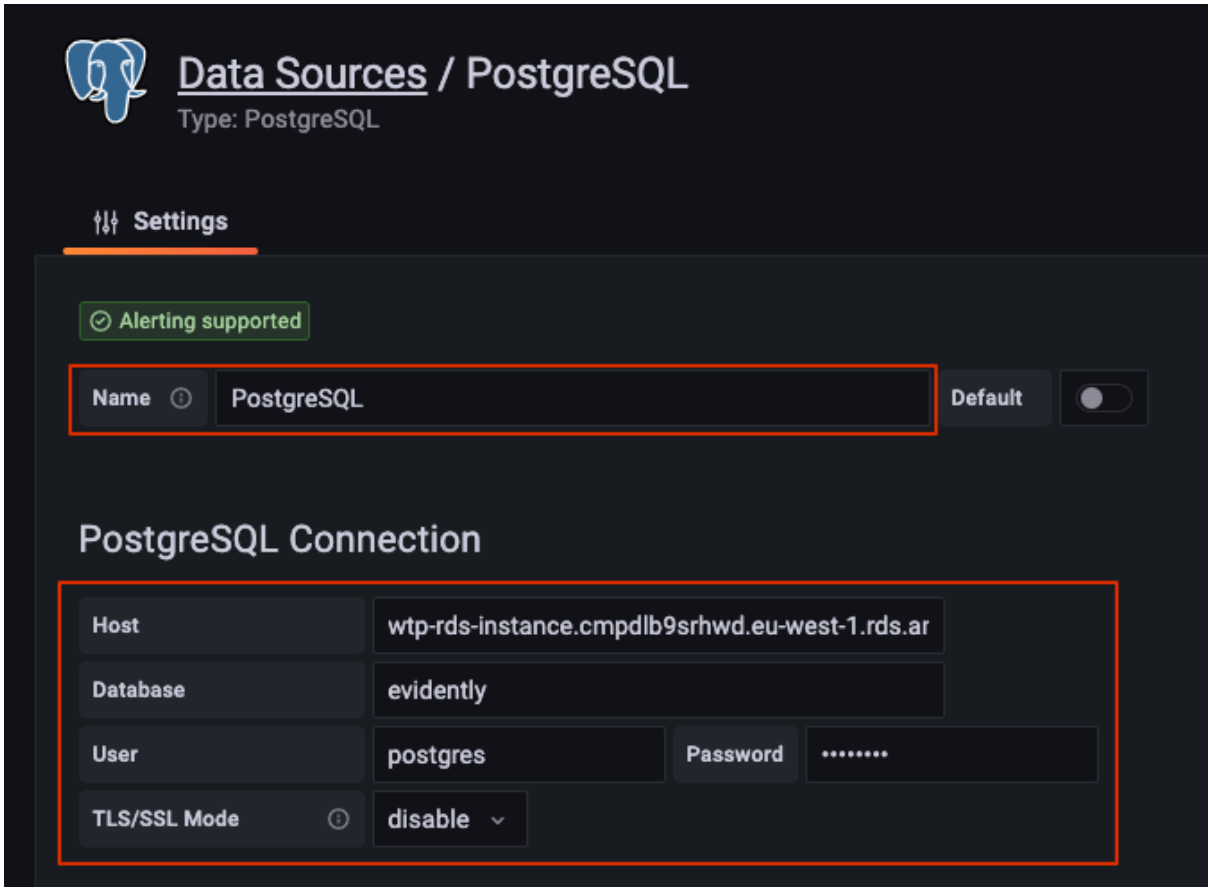



Exhibit-11: PostgreSQL Data Source Settings (Image by Author)

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,

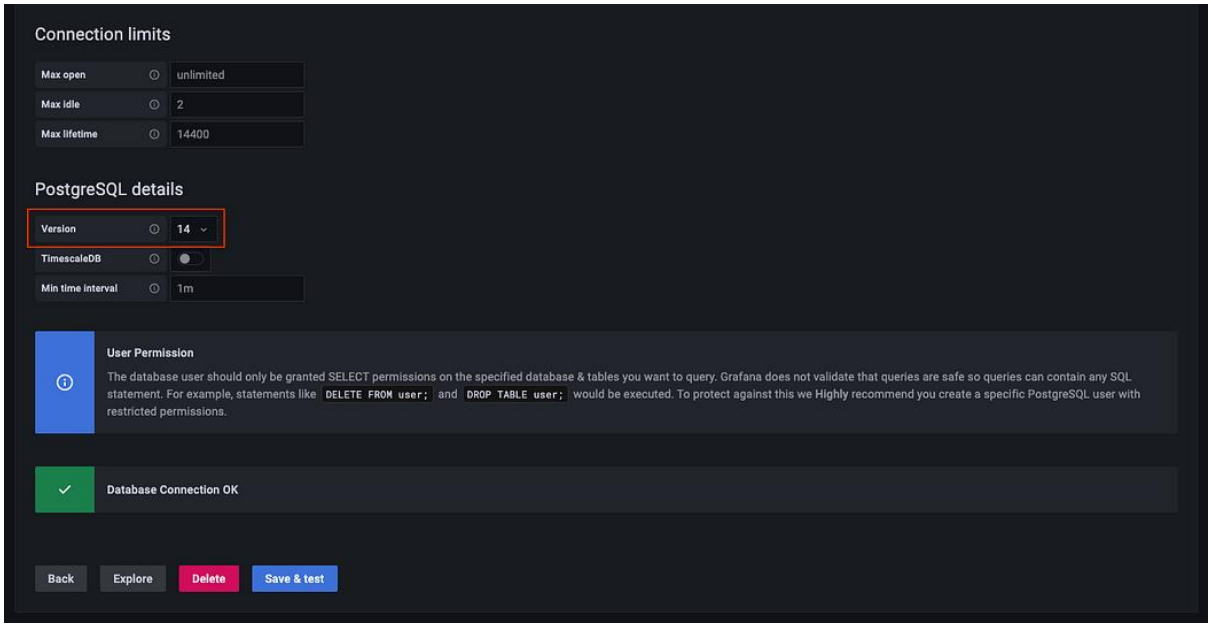



Exhibit-12: PostgreSQL Data Source Settings Continued (Image by Author)

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

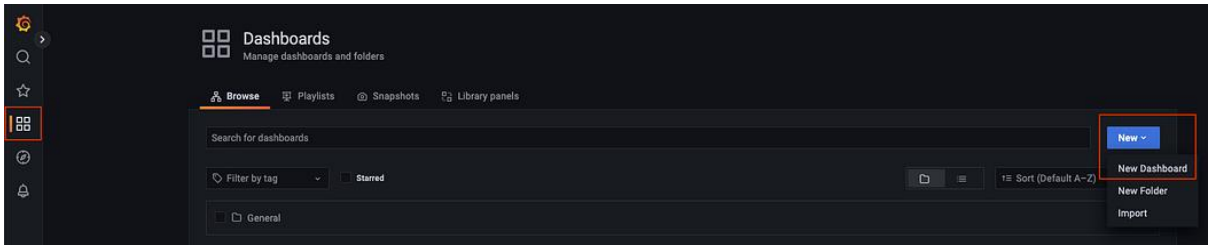



Exhibit-13: Creating a Grafana Dashboard (Image by Author)

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.

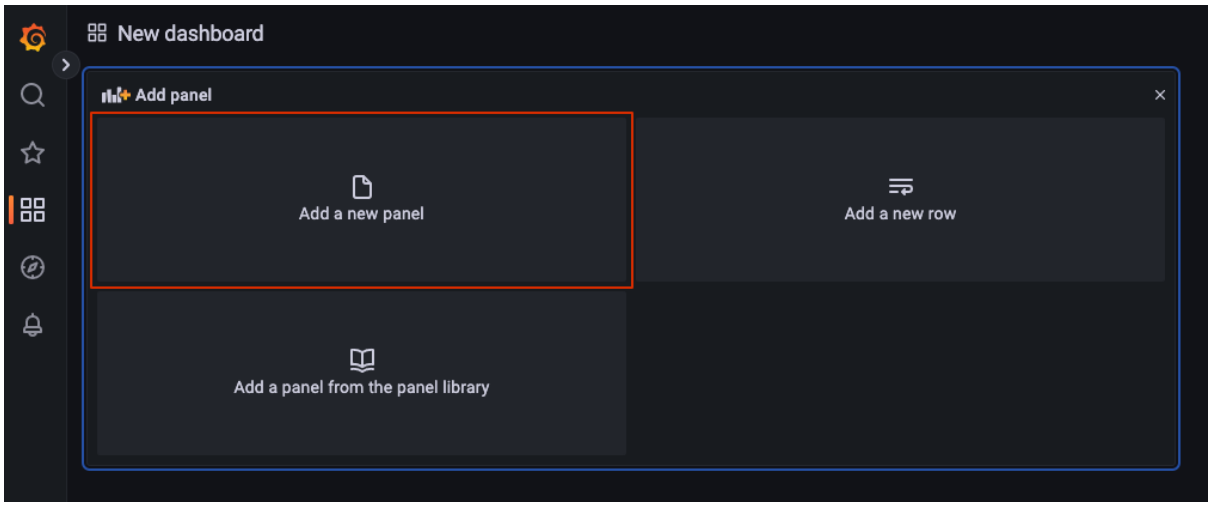


Exhibit-14: Adding a New Dashboard Panel (Image by Author)

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

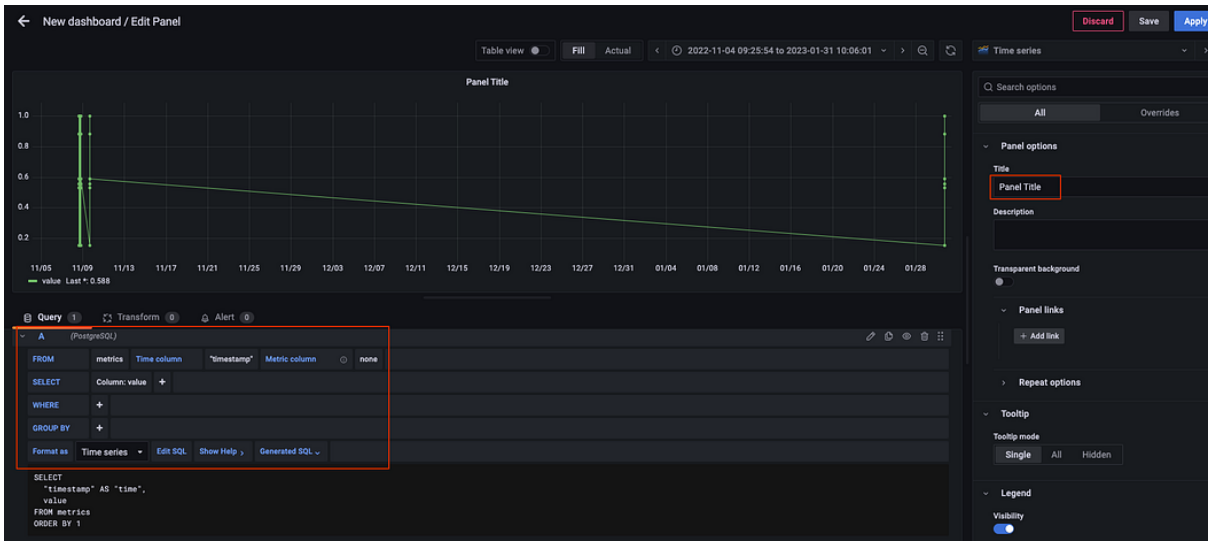



Exhibit-15: Query Statement for the Dashboard Panel (Image by Author)

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 | f9c1edbce7c9467d901446d0ce0364e7 | 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, timestampand 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:

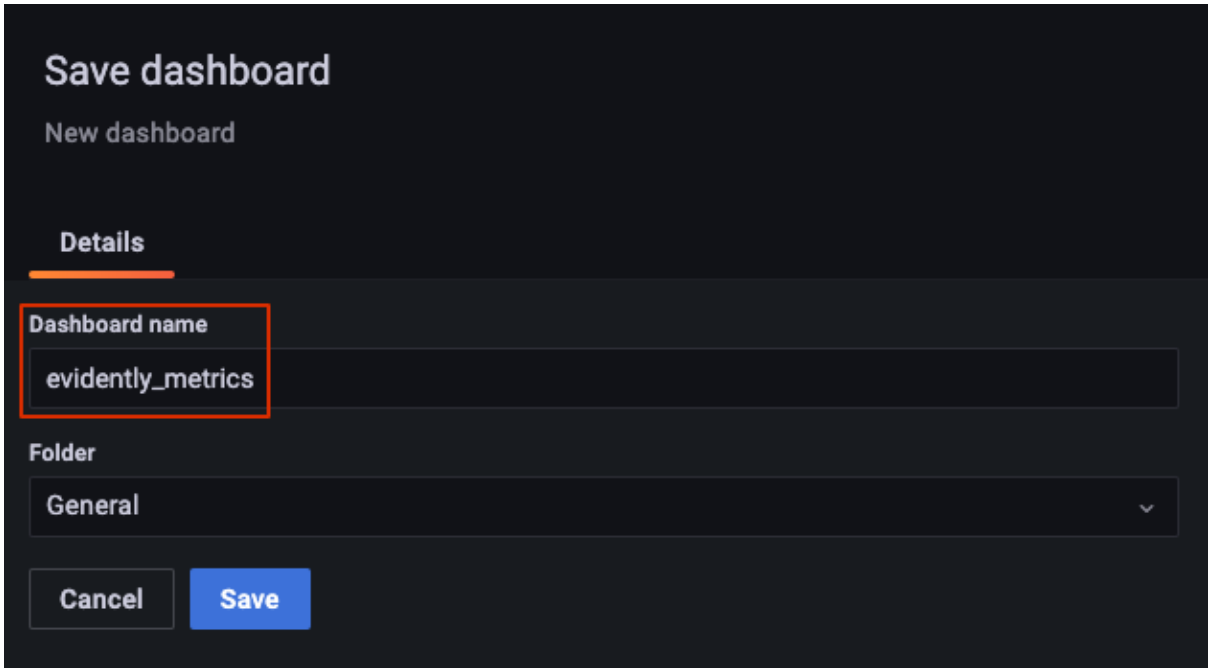



Exhibit-17: Saving the Grafana Dashboard (Image by Author)

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.

**Part 4: Developing, Scheduling, and Monitoring Batch-Oriented Workflows with Apache Airflow**

In the last part, we collected, stored, and visualized the ML model and data performance metrics with *Prometheus* and *Grafana*. Therefore, we have been able to finish all tasks in the flow of logic.

The flow of logic we showed in exhibits of the previous articles makes it easy for humans to understand where the operation starts and ends and what it does. However, it means nothing to computers. We need to convert the flow into a workflow that machines will understand. We achieve this by performing the tasks in the flow with *Python* functions. The function-task relationship here does not need to be one-to-one. We sort them by execution priority. In that way, we arrange all the functions as a process that, by the time it ends, will have fulfilled all tasks in the flow of logic and accomplished the project's goal.

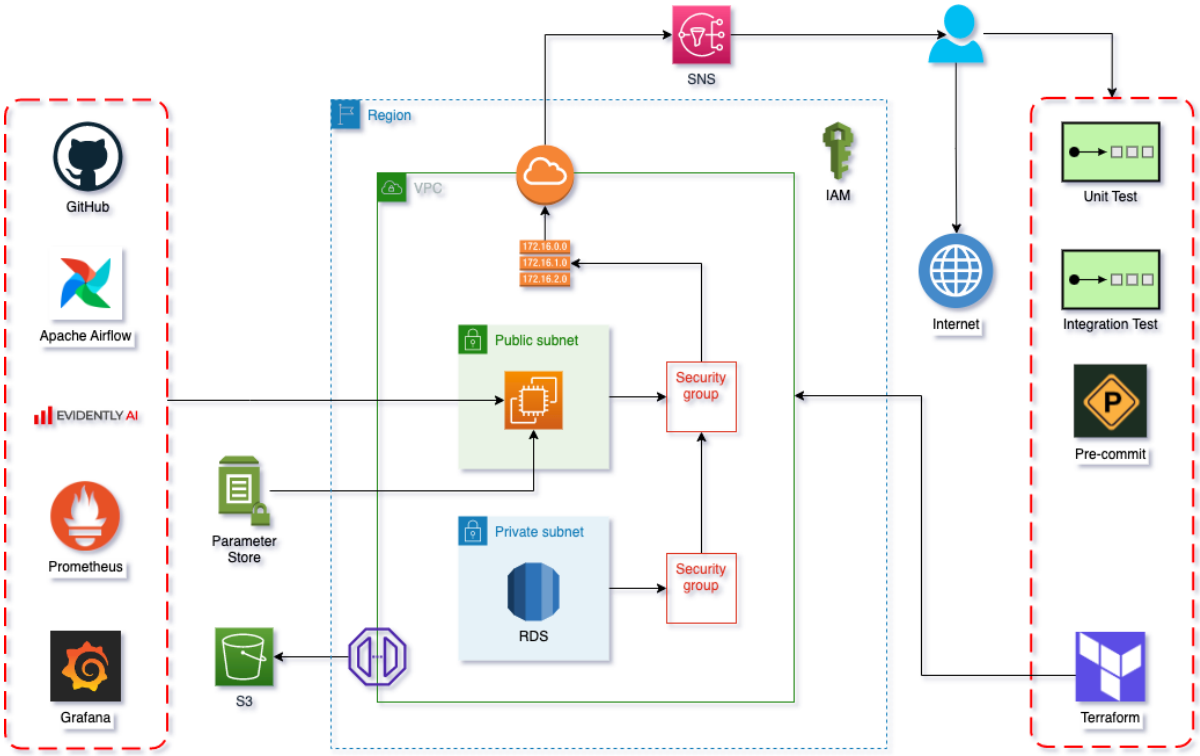
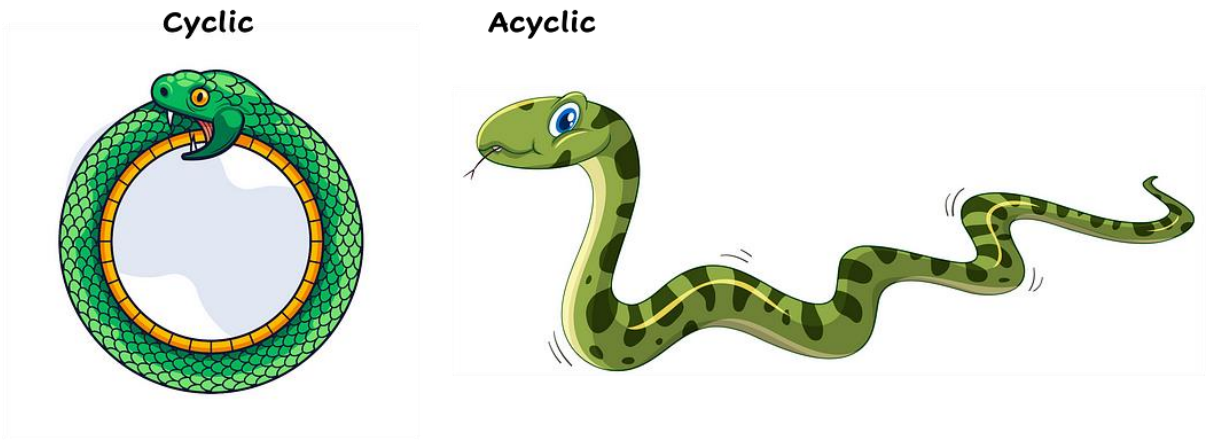



Exhibit-1: MLOps Project Diagram (Image by Author)

Briefly, the project periodically trains the model with the updated data to predict waiter tips, monitors its performance, and notifies the user about the results requested. Execution of functions follows the path like in a flowchart, starting with the first one and ending with the last. A function does not run unless all parent functions get executed. Such a process is, by nature, acyclic.

**Cyclic vs. Acyclic Graphs**

We need to spend a few more minutes to see the difference between cyclic and acyclic graphs before going further. "A cycle graph or circular graph is a graph that consists of a single cycle, or in other words, some number of vertices (at least 3, if the graph is simple) connected in a closed chain."[1] On the other hand, a graph with no cycles is acyclic.




Left Image by Freepik | Right Image by brgfx on Freepik | (Captions by Author)

Another concept is a “directed or undirected graph,” which can be cyclic or acyclic. A graph is called undirected if the edge between any two nodes has no particular direction, or the edge can be considered and identified as both from A to B and B to A where A and B are nodes or vertices. Depending on whether such a graph contains a cycle, it can be an Undirected Cyclic Graph (UCG) or Undirected Acyclic Graph (UAG), as shown in Exhibit 2.

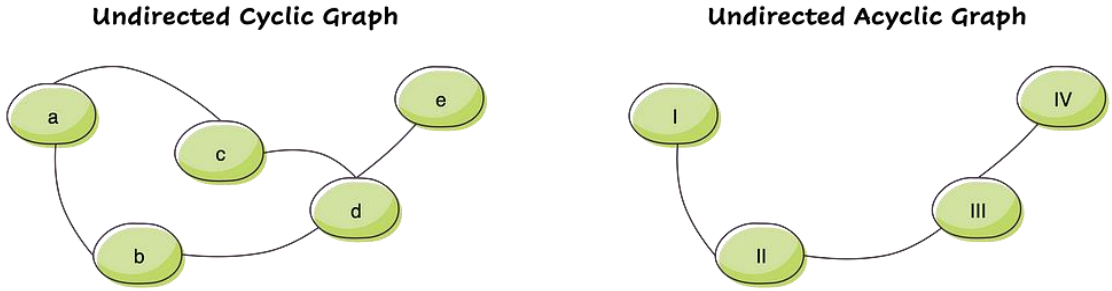



Exhibit-2: Undirected Graphs (Image by Author)

Conversely, if the edges have particular directions between nodes, as depicted in Exhibit 3, or the edge (A, B) is considered to be directed from A towards B but not the other way around, the graph is known as “directed.” Again, the inclusion or exclusion of cycles within the undirected graphs determines their types as Directed Cyclic Graph (DCG) or Directed Acyclic Graph (DAG).

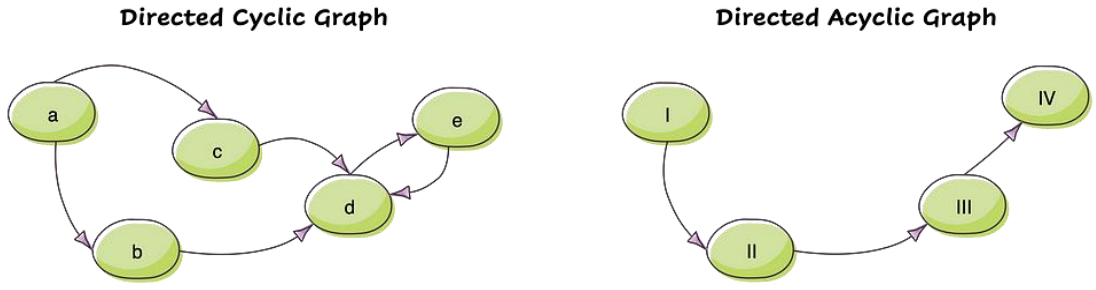



Exhibit-3: Directed Graphs (Image by Author)

To create our process, we need to use a DAG. Our DAG consists of nodes that are mainly *Python* functions. They are sorted in a particular order so that one does not get executed

until all preceding nodes run. This design should also allow us to schedule the entire run across the functions, detect any runtime error that might occur during the execution, perform retrials in case of failure, resume the execution from the failed step instead of starting over, see outputs and logs of a function, check its state during runtime, and maybe do even more. While even a genie grants three wishes only, who or what can deliver to us all these?! Though it is not so supreme, *Apache Airflow* can help us with them.

**An Introduction to Apache Airflow**

Airflow describes itself as :

*Apache Airflow is an open-source platform for developing, scheduling, and monitoring batch-oriented workflows. Airflow's extensible Python framework enables you to build workflows connecting with virtually any technology. A web interface helps manage the state of your workflows. Airflow is deployable in many ways, varying from a single process on your laptop to a distributed setup to support even the biggest workflows.[2]*

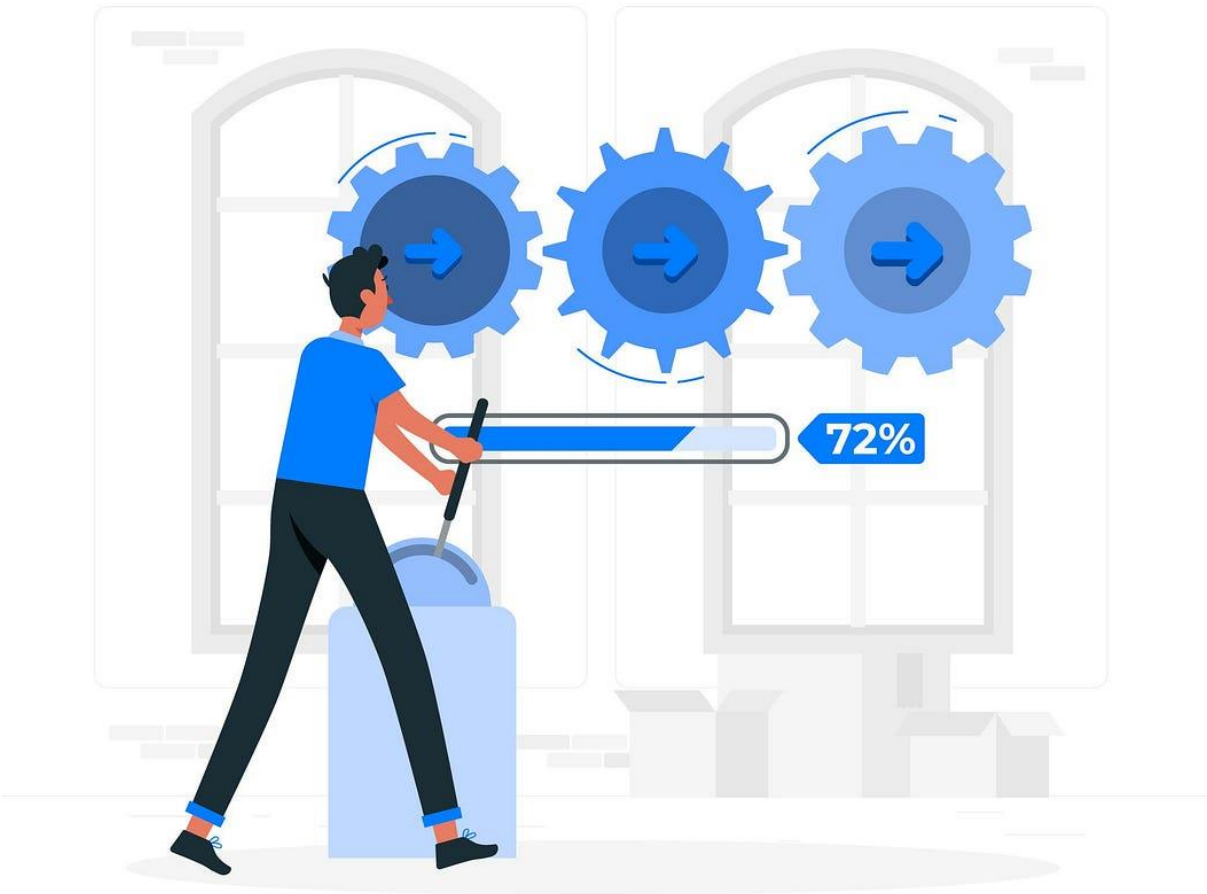



Image by storyset on Freepik

To avoid confusion in *Airflow* terminology, I will use function and task interchangeably, which means they both do the same thing.

*Airflow* represents a workflow with a DAG. Exhibit 4 shows the DAG of our project's tasks (functions):

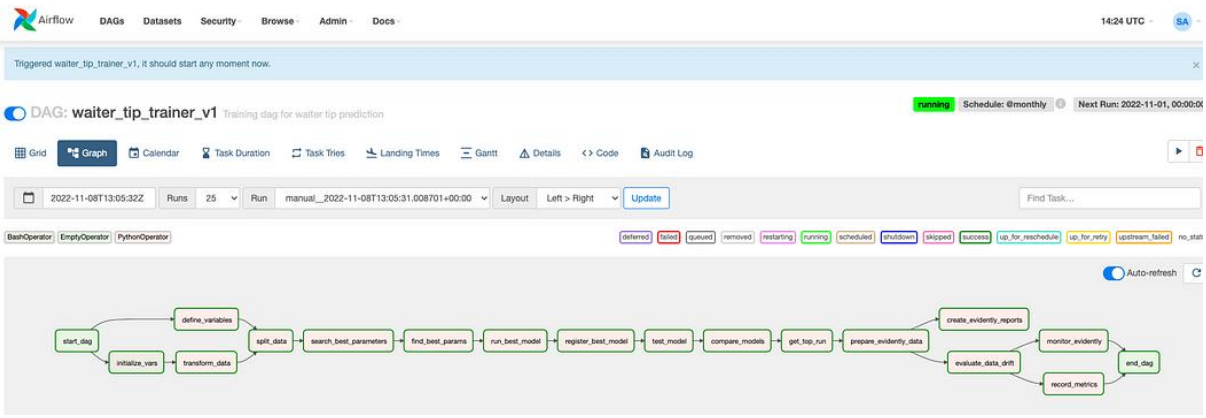



Exhibit-4: An Airflow DAG (Image by Author)

In this DAG, we see a directed and acyclic flow of tasks. By the default Trigger Rule, *Airflow* will run a task if and until all upstream ones run successfully. The Trigger Rule sets how *Airflow* should trigger a task. We can change the default behavior by picking any other of the Trigger Rules. Controlling the default behavior occurs by using the trigger_rule argument to a task.

Each rounded rectangle in the exhibit represents a task (function), and the dark green contour of a rectangle indicates successful completion. Right above the DAG on the right side, we see several rectangles with outlines of different colors. They each represent a specific state of tasks.

For example, light green indicates the task is running , and orange is upstream_failed , which means the preceding (upstream) one failed. The yellow one (up_for_retry) refers to retrying and rescheduling the failed task. The red outline means an error occurred during execution and failure to run followed after all retries with no success. One of 12 different states explains the current status of a task.

On the left side of the screen is the operator types (Python, Bash, Empty) legend. We'll later see what these operators are.

In the right corner at the top, the web UI shows the general status of DAG (running or none), the schedule interval, and the time of the next execution.

"A DAG Run is an object representing an instantiation of the DAG in time." DAG Runs are created according to the schedule interval that is a DAG argument. In other words, *Airflow* generates a DAG Run at each interval[3].

*Much in the same way that a DAG is instantiated into a DAG Run each time it runs, the tasks under a DAG are instantiated into Task Instances. An instance of a Task is a specific run of that task for a given DAG (and thus for a given execution_date). They are also the representation of a Task that has a state, representing what stage of the lifecycle it is in.[4]*

While the DAG is in running mode, we observe that each task instance changes line colors, indicating it transitions from one state to another. This execution follows a routine procedure across states. A typical cycle of tasks runs in the following way, as depicted in Exhibit 5:

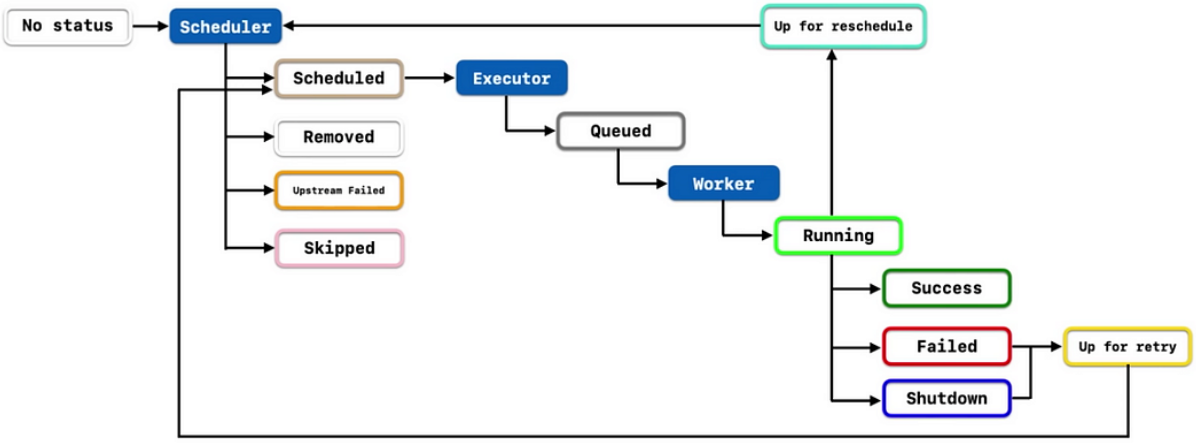



Exhibit-5: The Airflow Task Lifecycle (Taken from https://youtu.be/K9AnJ9_ZAXE)

Therefore, the successful completion of a task on the first attempt goes through five steps: none , scheduled , queued , running , and success.

When we click each task, we can see the logs and other relevant information for that task instance. One can further explore the UI by navigating the lower menu (Grid, Calendar, Task Duration, etc.) I leave that experience with the curious developer and present the following display to give some idea about the basic architecture of *Airflow*.

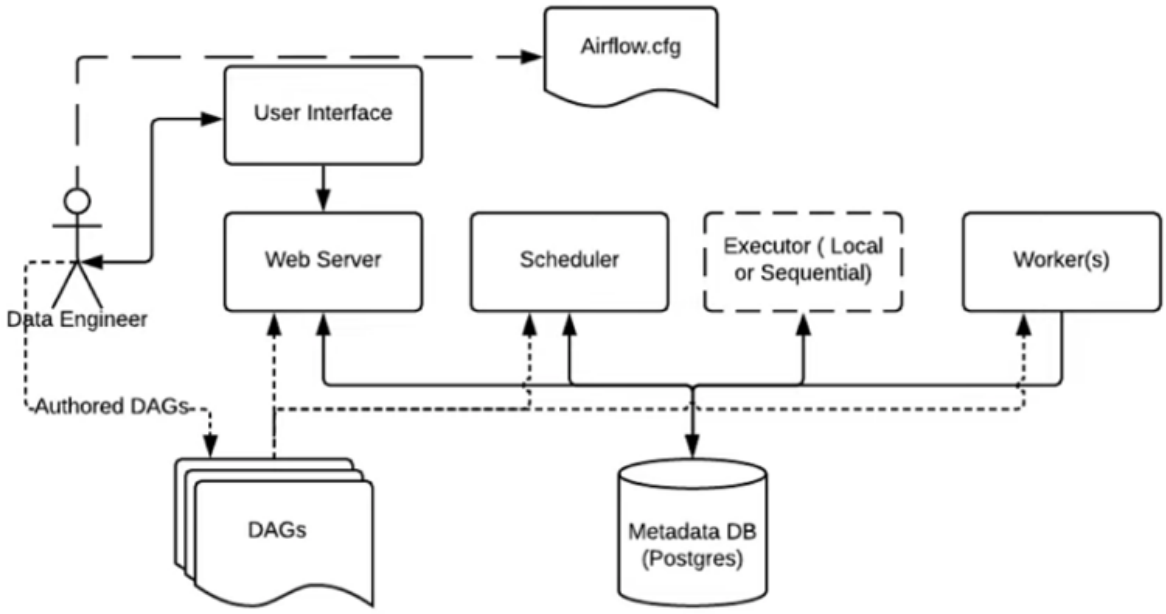



Exhibit-6: The Basic Architecture of Airflow (Taken from https://youtu.be/K9AnJ9_ZAXE)

Although we are not going into explaining the components shown in Exhibit 6 immediately, it will become clear what they are and how they work when we encounter them throughout the article.

It has been an overview of what *Airflow* is and how it works. Now, we can proceed to install and code for it.

**Installation and Configuration**

We install *Apache Airflow* (2.3.3):

```
pip install 'apache-airflow==2.3.3' \
 --constraint "https://raw.githubusercontent.com/apache/airflow/constraints-2.3.3/constraints-3.9.txt"
```

We use this installation method to install *Apache Airflow* on physical or virtual machines. The package manager pip is the only officially supported mechanism of this installation using constraint mechanisms.

*The constraint files are managed by* Apache Airflow *release managers to make sure that you can repeatably install* Airflow *from PyPI with all Providers and required dependencies.[5]*

We need to install compatible *Airflow* and *Python* versions. Therefore *Airflow* dependencies should be working for the *Python* version. We make sure that happens by adding the constraint argument in the pip installation command. This argument has the following format:

```
https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERSION}/constraints-${PYTHON_VERSION}.txt
```

We have to specify the correct *Airflow* and *Python* versions in the URL. If we type the URL https://raw.githubusercontent.com/apache/airflow/constraints-2.3.3/constraints-3.9.txt on our browser, we see the list of dependencies that work with both packages. In our installation, *Airflow* 2.3.3 and *Python* 3.9 seem to be compatible. They have already worked successfully on Mac and Linux.

We check the *Airflow* version after the installation:

```
airflow version
```

After the installation, *Airflow* creates a folder named “airflow” and places configuration, database, and log files in it. We do probably not want to work on it. Instead, we’ll create another folder inside our application directory and change the *Airflow* path variable to its new address that contains this folder:

```
export AIRFLOW_HOME=/home/ubuntu/app/Waiter-Tips-Prediction
```

This way, we move the *Airflow* home directory to the current directory. Above command will collect all *Airflow* files within the “Waiter-Tips-Prediction” folder when we run the DAG. In doing so, we should use the absolute path for the current directory as we did. We need to execute this command every time we start the server. To automate this minor operation without having to repeat it every time at launch, we open the .bashrc file:

```
nano /home/ubuntu/.bashrc
```

and append the following line to the file:

```
export AIRFLOW_HOME=/home/ubuntu/app/Waiter-Tips-Prediction
```

The .bashrc file is a script file executed when a user logs in. It contains the necessary setting up or enabling configurations for the terminal session. After saving and exiting the file, we can log out of the session and log back in or execute the following command to reflect the changes:

source .bashrc

Then, we have to initialize the *Airflow* database in the project folder (/app/Waiter-Tips-Prediction). The following command will create an SQLite database for logs and configuration files (we'll do this on RDS, however):

airflow db init

To use the web UI and log into the database, we should create a user:

airflow users create --username ${USERNAME} --password ${PASSWORD} --firstname ${FIRSTNAME} --lastname ${LASTNAME} --role ${ROLE} --email ${EMAIL}

We set a username and password and enter the first and last names. The role can be Admin, and email doesn't need to be genuine. Setting up credentials is one-time only.

Next, we connect to the *Airflow* server to use the web UI:

airflow webserver -p 8080 -D

The *Airflow* server works in the detached mode and listens at port 8080. We also need to start the scheduler:

airflow scheduler -D

The *Airflow* scheduler works in the detached mode too. The *Airflow* scheduler monitors all DAGs and tasks and triggers the task instances if their dependencies are in place. It monitors and gets synchronized with the project folder (/app/Waiter-Tips-Prediction) where DAG files reside and regularly checks if any active tasks exist to trigger. The *Airflow* scheduler runs as a permanent service and uses the configuration specified in the airflow.cfg file[6].

Now, we can open the web UI by typing http://0.0.0.0:8080 in the browser. Since *Airflow* is on EC2, the page opens on the local machine after port forwarding. The opening page will prompt us to enter our username and password:

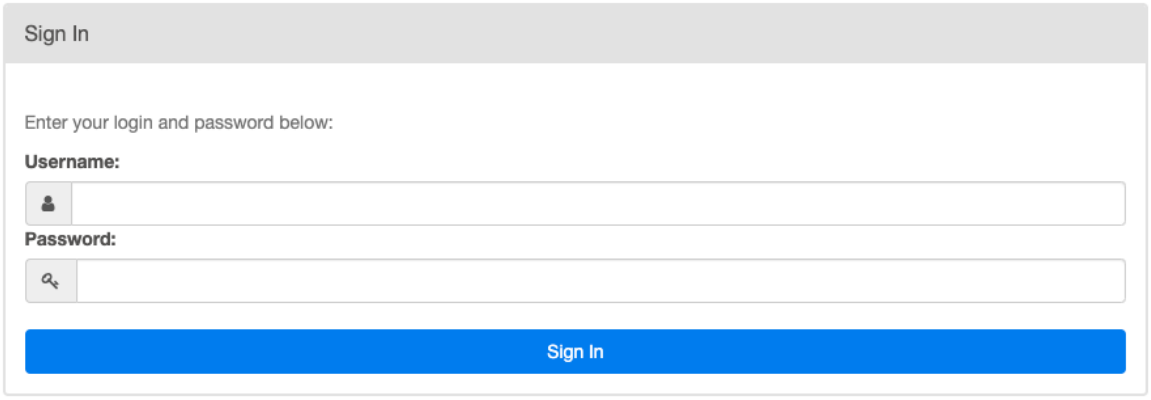



Exhibit-7: Airflow Login (Image by Author)

It will only ask you once to enter credentials and lets you get in automatically when you open the browser the next time. After login, we will notice several DAG examples on the page, but not our DAG! We may want to discard those examples and instead want to see our DAG. For this, we need to make some changes in the configuration file, airflow.cfg. The configuration file also contains database information. That is the metadata database where the details about *Airflow* runs are stored. Our RDS serves *Airflow* to achieve this purpose as well as it does for *MLflow* and *Evidently*.

The airflow.cfg file is located inside the project folder. The first change we have to make is to set the value to False for the load_examples argument:

```
# Are DAGs paused by default at creation
dags_are_paused_at_creation = True

# The maximum number of active DAG runs per DAG. The scheduler will not create more DAG runs
# if it reaches the limit. This is configurable at the DAG level with ``max_active_runs``,
# which is defaulted as ``max_active_runs_per_dag``.
max_active_runs_per_dag = 16

# Whether to load the DAG examples that ship with Airflow. It's good to
# get started, but you probably want to set this to ``False`` in a production
# environment
load_examples = True ——————→ Set this value to 'False' to discard DAG examples
```

Exhibit-8: DAG Examples in Airflow Configuration File (Image by Author)

The second adjustment is to change the executor value to LocalExecutor :

```
# The executor class that airflow should use. Choices include
# ``SequentialExecutor``, ``LocalExecutor``, ``CeleryExecutor``, ``DaskExecutor``,
# ``KubernetesExecutor``, ``CeleryKubernetesExecutor`` or the
# full import path to the class when using a custom executor.
executor = LocalExecutor
```

Exhibit-9: Executor in Airflow Configuration File (Image by Author)

The default executor is SequentialExecutor. In LocalExecutor, task instances get executed as subprocesses.

The third change is to set the endpoint of the PostgreSQL database for metadata:

```
[database]
# The SqlAlchemy connection string to the metadata database.
# SqlAlchemy supports many different database engines.
# More information here:
# http://airflow.apache.org/docs/apache-airflow/stable/howto/set-up-database.html#database-uri
# sql_alchemy_conn = sqlite:////Users/hasanserdaraltan/Downloads/waiter_tips_prediction/airflow.db
sql_alchemy_conn = postgresql+psycopg2://postgres:password@wtp-rds-instance.cmpdlb9srhwd.eu-west-1.rds.amazonaws.com:5432/airflow
```

Exhibit-10: SQL Connection in Airflow Configuration File (Image by Author)

Here, we type the endpoint of our RDS. As you may notice, we have created a new database in RDS for *Airflow,* "airflow." We will see how to create databases in RDS later. Our endpoint also contains our username and password. We should execute airflow db init command after

updating the SQL endpoint in the configuration file if we want to use RDS as the metadata database.

The last modification, which may or may not be necessary, is about the DAG folder location. After installing *Airflow* in the project folder and updating the path variable, we should see it decides the location itself, yet, we need to make sure it is correct:

```
[core]
# The folder where your airflow pipelines live, most likely a
# subfolder in a code repository. This path must be absolute.
dags_folder = /home/ubuntu/app/Waiter-Tips-Prediction/dags

# Hostname by providing a path to a callable, which will resolve the hostname.
# The format is "package.function".
#
# For example, default value "socket.getfqdn" means that result from getfqdn() of "socket"
# package will be used as hostname.
#
```

Exhibit-11: DAG Folder Location in Airflow Configuration File (Image by Author)

It seems that the DAG folder is specified correctly. Now, at this point, you may have realized that we haven't created any DAG folder yet. Within the project folder (/app/Waiter-Tips-Prediction), we create a folder named "dags," and within the "dags" folder, all subfolders where scripts that contain *Airflow* tasks (functions) reside. The hierarchy of the directory and files should be similar to the partial representation in the following exhibit:

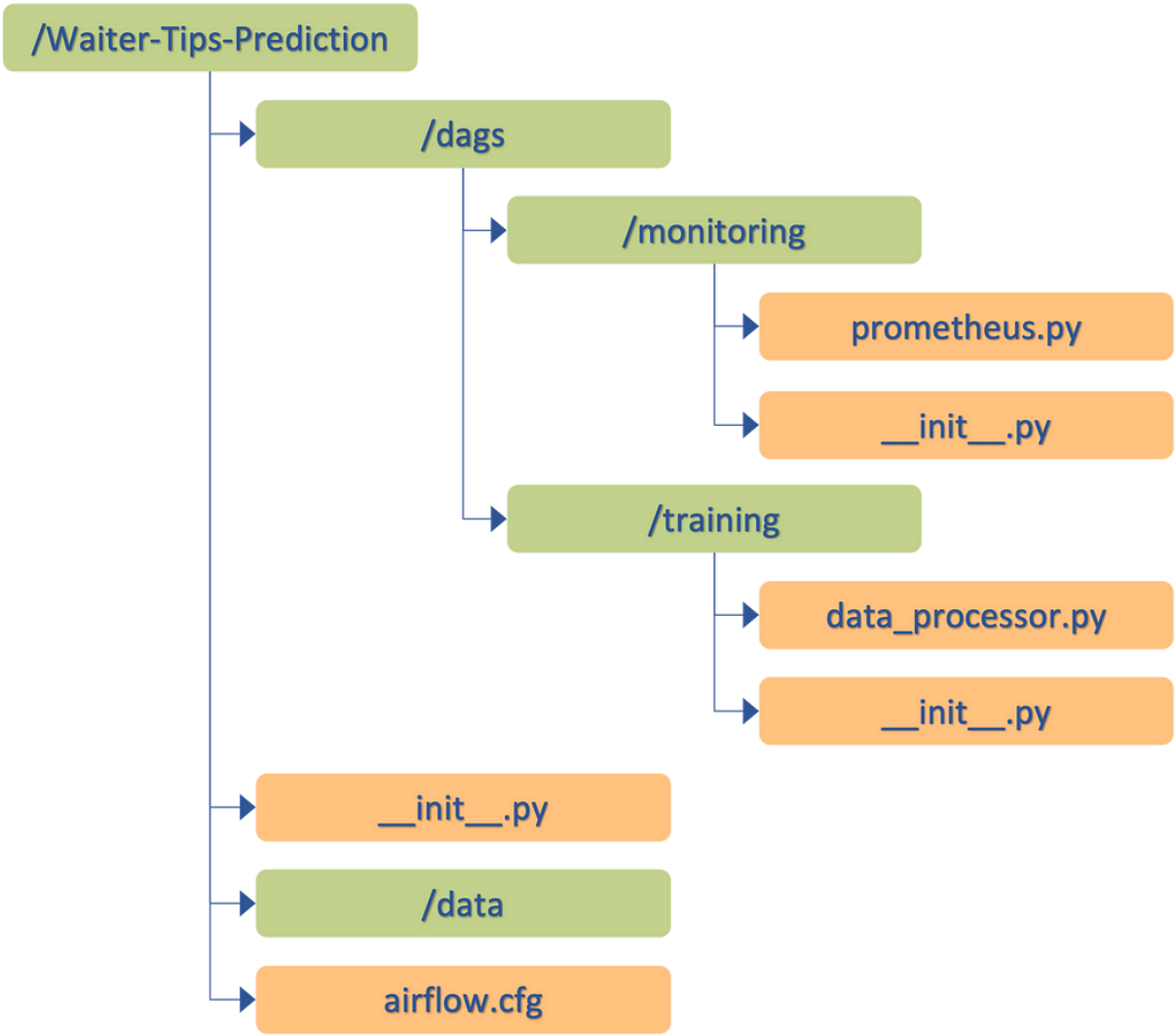



Exhibit-12: Simplified Version of the Project Folder Hierarchy (Image by Author)

All modules that execute the tasks displayed in Exhibit 4 should be under the “dags” directory. We can spread them across subfolders under the “dags” directory to organize the application objects. There should be a __init__.py file within the “dags” folder and every subfolder. One may refer to the GitHub page to see the entire hierarchy of the dags folder.

Important to note that if we want to use RDS as the database, we first need to make the above changes in the configuration file, create a dags folder and then launch the database, create a user, start the scheduler by executing the following commands, and then open the web UI and enter our credentials:

```
airflow db init
airflow users create --username ${USERNAME} --password ${PASSWORD} --firstname
${FIRSTNAME} --lastname ${LASTNAME} --role ${ROLE} --email ${EMAIL}
airflow webserver -p 8080 -D
airflow scheduler -D
```

When we open the browser at http://0.0.0.0:8080 the next time, *Airflow* will not ask us to enter the credentials again and show the DAG examples anymore. Instead, we’ll proudly see our project DAG!

Every time we terminate the session and come back to run the *Airflow* workflow, we need to execute the following commands in order on the project folder's terminal:

```
airflow db init
airflow webserver -p 8080 -D
airflow scheduler -D
```

The first command initializes the database, the second command launches the server, and the third command starts the scheduler. I strongly recommend you check if the server daemon and scheduler are working after the launch:

```
lsof -i tcp:8080
lsof -i tcp:8793
```

The first line inspects the server, and the second line checks the scheduler. It happened that I had to restart the *Airflow* server and scheduler to let the changes take effect or remove any errors. We will discuss these issues in Troubleshooting.

We have finished the installation and configuration and can get to coding for *Airflow*.

**Coding for Airflow**

We are going to discuss three components of *Airflow*: "DAG," "XComs," and "Variables." We'll start with DAG.

**DAG**

To execute a workflow as in Exhibit 4, first, we need to create a coordinating script such as airflow_dag.py within the "dags" directory. Let us dive into this file:

```
# Import libraries
from datetime import datetime, timedelta

from airflow.models import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.empty import EmptyOperator
from airflow.operators.python import PythonOperator
from monitoring.evidently_monitoring import (
    create_evidently_reports,
    evaluate_data_drift,
    monitor_evidently,
    prepare_evidently_data,
    record_metrics,
)
from training.data_preprocessor import split_data, transform_data
from training.data_processor import (
    compare_models,
    find_best_params,
    get_top_run,
    register_best_model,
    run_best_model,
    search_best_parameters,
```

```
    test_model,
)
from training.var_init import define_variables
```

We import the DAG module and operators from *Airflow*. We have already seen the other functions and modules before. We discussed the Monitoring and Training modules and their methods when we saw *MLflow*, *Evidently*, and *Prometheus*. Exhibit 12 displays the partial representation of these modules. Note that all the functions (tasks) contained in these modules are the same as shown in Exhibit 4. For a closer look, I'll present its cropped version here below:

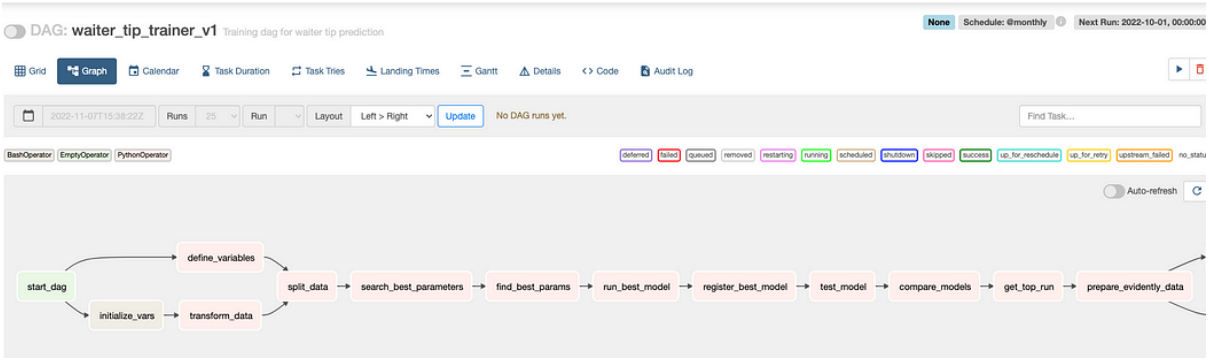



Exhibit-13: Airflow DAG Tasks (Image by Author)

What we see in Exhibit 13 is the functions of the Monitoring and Training modules. Next, we define the environment variables we'll need throughout the application. Again, we have seen them already in the earlier scripts.

```
MLFLOW_EXPERIMENT_NAME = "mlflow-experiment-1"
EVIDENTLY_EXPERIMENT_NAME = "evidently-experiment-1"
MODEL_NAME = "xgboost-model"
BUCKET_NAME = "s3b-tip-predictor"
KEY = "data/tips.csv"
CURRENT_DIR = "/home/ubuntu/app/Waiter-Tips-Prediction/"
TEST_SIZE = 0.2
TAG_MLFLOW = "xgboost"
TAG_EVIDENTLY = "evidently"
METRIC = "metrics.rmse ASC"
MAX_RESULTS = 5000
```

While tracking experiments of the model and *Evidently* tests, we ran them under respective names MLFLOW_EXPERIMENT_NAME and EVIDENTLY_EXPERIMENT_NAME. We used MODEL_NAME to name the model in *MLflow* runs. BUCKET_NAME and KEY make up the path to our S3 bucket and folder. CURRENT_DIR is the project folder's path. We use TEST_SIZE in splitting the pre-processed data. TAG_MLFLOW and TAG_EVIDENTLY are for tagging runs in *MLflow* experiments. METRIC is the criterion used by *MLflow* to sort experiments and runs up to MAX_RESULTS , which is 5000 in number.

Next, we need to set the *Airflow* arguments:

```
default_args = {
    "owner": "serdar",
    "start_date": datetime(2022, 8, 25, 2),  # days_ago(2)
```

```
    "end_date": datetime(2022, 12, 25, 2),
    "depends_on_past": False,
    "email": ["serdar@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(seconds=10),
}
```

The argument owner can be you, a group of developers, or any entity that owns the DAG. The scheduler examines the lifetime of the DAG, which is from the start_date to the end_date, and creates the DAG Run based on the specified interval (once, hourly, daily, weekly, etc.) sequentially[6].

We set depends_on_past to True if we need a task to run if its previous run in the earlier DAG Run succeeded[7]. I prefer to declare it as False unless a reason exists to do otherwise.

The argument retries refers to how many times *Airflow* should try the task before designating it as Failed. Though I set it as 1 here, assigning a higher threshold is wise.

Another parameter is retry_delay , which sets the time in seconds that should lapse between consecutive retries.

We may let *Airflow* notify the email owner when a task is retried (email_on_retry) or failed (email_on_failure) by setting their values to True. In more complex workflows, email alerts can be necessary upon retry or failure, but for a project like ours, I don't see a reason not to set them as False.

Default arguments hold for all tasks. However, we can override some of them within the task configuration, such as start_date.

There are three alternatives for declaring a DAG. We will use a context manager, which will implicitly add the DAG to anything inside it. As seen below, we list, as an example, only three of the tasks within the DAG context, which employ three different Operators: Empty , Python , and Bash. All tasks use an operator. "An Operator is conceptually a template for a predefined Task that you can just define declaratively inside your DAG.[8]"

```
with DAG(
    dag_id="waiter_tip_trainer_v1",
    default_args=default_args,
    description="Training dag for waiter tip prediction",
    schedule_interval="@monthly",  # "*/13 * * * *",
    dagrun_timeout=timedelta(minutes=60),
    catchup=False,
    tags=[TAG_MLFLOW],
) as dag:

    task_start_dag = EmptyOperator(
        task_id="start_dag",
    )
```

```
task_define_variables = PythonOperator(
    task_id="define_variables",
    python_callable=define_variables,
    op_kwargs={
        "mlf_dict": {
            "mlflow_experiment_name": MLFLOW_EXPERIMENT_NAME,
            "evidently_experiment_name": EVIDENTLY_EXPERIMENT_NAME,
            "model_name": MODEL_NAME,
        },
        "s3_dict": {"bucket_name": BUCKET_NAME, "key": KEY},
        "local_dict": {"current_dir": CURRENT_DIR},
    },
    do_xcom_push=False,
)

task_initialize_vars = BashOperator(
    task_id="initialize_vars",
    bash_command="sleep 10",
    do_xcom_push=False,
)
```

We may create more than one DAG in our application. We identify the current DAG with a dag_id , tag it with tags , and add a description.

The parameter default_args takes as input the default arguments we defined above.

As earlier explained, schedule_interval indicates at what intervals *Airflow* runs the DAG between the start_date and end_date. We set it as monthly here, which means our DAG will run once a month. One can refer to the *Airflow* documentation for more information about "schedule intervals."

We specify how long a DAG Run should be up before timing out or failing with the dagrun_timeout parameter. Only scheduled DAG Runs enforce the timeout.

The scheduler catchup is, by default, True , and denoted as catchup_by_default in the configuration file (airflow.cfg). We can set this to False in the said file or override it in the DAG context as above. When it is True , the scheduler will create a DAG Run for each completed interval between start_date and the time that the scheduler daemon picks up the DAG, and the scheduler will execute them sequentially. Let us suppose now is 2022–11–25 at 2:01 AM. When we initiate the DAG Run for the first time, it will make three runs, one for each of the past completed intervals from 2022–08–25 at 2:00 AM if catchup is set to True. So, the scheduler catches up with the schedule! The number of runs will increase depending on the value of the schedule interval parameter. If it were daily , the scheduler would run the DAG 93 times for the past completed intervals. Conversely, if catchup is set to False , the scheduler runs the DAG once only on 2022–11–25 at 2:01 AM and executes the next run on 2022–12–25 at 2:00 AM.

Now, we have created a DAG instance named "dag." The first operator we'll place inside it is EmptyOperator. EmptyOperator is a dummy operator and does nothing literally. It establishes the starting point of the workflow and names it as start_dag.

The next task (id: define_variables) uses PythonOperator. PythonOperator calls an arbitrary *Python* function (define_variables). We use define_variable as both the task id and the callable name. We don't have to do this, though. Any name is good for the task id. The DAG diagram (see Exhibit 4) displays the tasks by their task id.

A PythonOperator task has the python_callable , which, as a value, has the name of a particular function from the Monitoring or Training modules. That means this PythonOperator task will call and execute that *Python* function.

In the task configuration, we see another argument, op_kwargs , which consists of three dictionaries (mlf_dict, s3_dict, local_dict) with their values composed of the environment variables we defined before. That means it inputs these three dictionaries into the define_variables function as arguments while calling it. In turn, this function will run with the dictionaries as arguments. Indeed, the first few lines of define_variables look as follows:

```
def define_variables(
    mlf_dict: dict, s3_dict: dict, local_dict: dict
) -> tuple[str, str, str]:

    """
    Takes new experiment and/or model variables
    as user input or allows to continue with the
    current ones. Sets the S3 and local path
    variables. Registers all of them in Airflow.
    """

    date_time = datetime.now().strftime("%Y-%m-%d / %H-%M-%S")
    logging.info("Date/time '%s' is set.", date_time)

    # User input
    mlflow_experiment_name_input = mlf_dict["mlflow_experiment_name"]
    evidently_experiment_name_input = mlf_dict["evidently_experiment_name"]
    model_name_input = mlf_dict["model_name"]
```

That is one of the ways for tasks to get their input. We are going to see do_xcom_push=False when we discuss XComs.

Another operator we use is BashOperator , which executes a bash command. The BashOperator executes a bash_command: "sleep 10 ". It delays the execution for 10 seconds. We put it there to avoid a racing issue between define_variables and transform_data because the latter needs the output of the former. The BashOperator provides define_variables with enough time to wrap up. We could have arranged these two functions sequentially instead of using BashOperator. That is right! I just wanted to see how BashOperator works:)

The default behavior for triggering tasks in the dag file is all_success. This option dictates that all upstream jobs must succeed for a task instance to run. For example, we may add another task and set its trigger_rule as all_failed. It will be triggered when all parent tasks are in a failed or upstream_failed state. Therefore, we can let it send us an alert, for example, in case all upstream jobs have failed.

The other operators/tasks in the dag file are similar and contain the same arguments yet with different values. A crucial question has remained still unanswered: How does *Airflow* know the order of these tasks?

Two ways exist to declare individual task dependencies. The recommended one is to use the >> and << operators[7]. As Exhibit 4
shows, start_dag, initialize_vars, define_variables, transform_data, split_data run in parallel or sequentially. We named them in the DAG
as task_start_dag , task_initialize_vars , task_define_variables, and so on. Our DAG knows the task instances by these names. All we need to do is to sort the task instances with the >> operator. For example:

```
task_start_dag >> [task_define_variables, task_initialize_vars]
task_initialize_vars >> task_transform_data
[task_define_variables, task_transform_data] >> task_split_data
```

The relationship map in the fragment above tells our DAG to execute task_start_dag first, then task_define_variables and task_initialize_vars in parallel, and then go on with task_transform_data once task_initialize_vars is complete, and lastly, run task_split_data after the execution of task_define_variables and task_transform_data. We use brackets to specify the task instances that can run in parallel. The left side of the >> operator is the upstream job, while the right part is the downstream job. We can break down the map into several lines by repeating the last task of the above line at the beginning of the current line.

Finally, we have finished the DAG component. You can find the code in one piece here. Now, we move on to XComs.

**XComs**

In *Python*, we usually exchange data between functions by declaring a return statement in the supplier function and calling it from the client function. *Airflow* doesn't work that way. *Airflow* tasks are, by default, isolated pieces and may be running on a distributed network[9].

XComs come to our help to facilitate the exchange. XComs is the short version of "cross-communications." It sounds, to me, like the name of a secret department in an intelligence organization! Now, let us get into a bit more detail by seeing it in action:

```
def find_best_params(ti: Any, metric: str, max_results: int) -> dict[str, Any]:

    """
    Retrieves the parameters of the best model run of a particular experiment
    from the mlflow utils module.
    """

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

    # Get the best params from mlflow server
    best_params = get_best_params(mlflow_client, experiment_name, metric, max_results)
```

```
    # Push the best params to XCom
    ti.xcom_push(key="best_params", value=best_params)

    return best_params
```

In this code, we compute and get best_params and push it to XCom storage with the xcom_push command on the task instance. ti stands for task instance and enters the function as an argument. We store the best_params as a value with the key best_params. XCom adds a timestamp also:

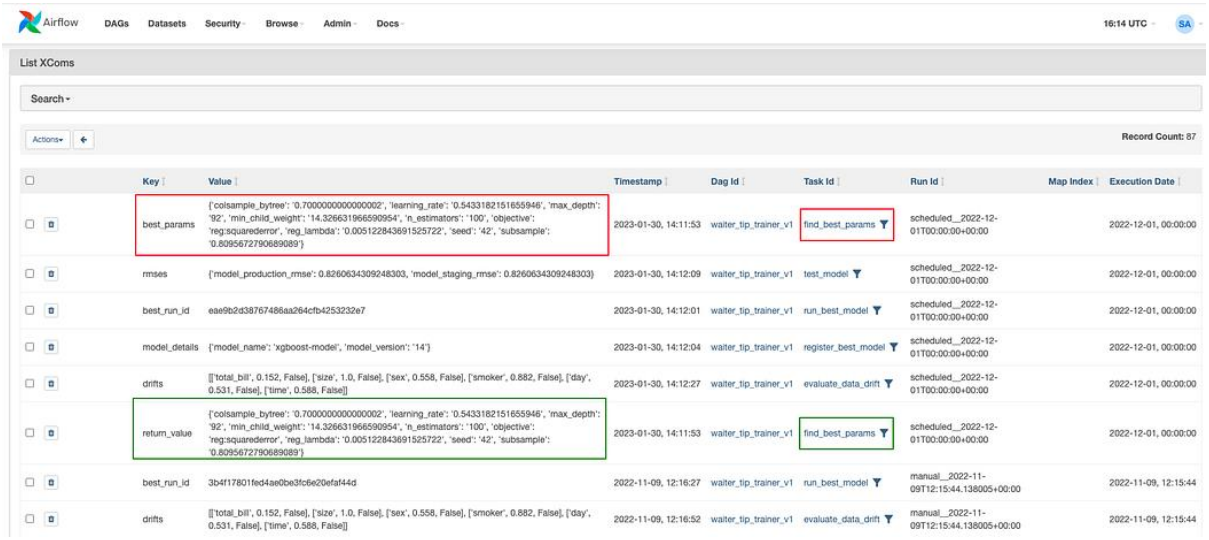



Exhibit-14: Airflow XComs (Image by Author)

We can access XComs on the web UI by clicking first “Admin” and then “XComs” from the top menu.

Exhibit 14 shows us that the best_params is pushed from the find_best_param task and stored as a key-value pair in its respective columns in the list. Many operators will automatically send their results into an XCom key called return_value if the do_xcom_push argument is set to True as it is by default, and the operator contains the return statement[9].

We have set do_xcom_push to False in several jobs in the dag file above, so they didn’t push anything to XCom. Tasks that don’t have the do_xcom_push argument and don’t have the return statement and xcom_push in their codes neither return a result nor push anything to XCom. However, I made a mistake here: In the DAG context, I did not mention do_xcom_push (then, it was True by default) for the find_best_param task, and I used *both* xcom_push *and* put the return statement in the function. That created a duplicate record in XCom. In addition to the data generated by xcom_push with the key best_params (red box), I made another record with the absence of do_xcom_push and the presence of the return statement (green box). As a result, we have ended up with the freedom to choose either one as input later!

Exhibit 14 also shows the other values pushed to XCom from various task instances. Anyone may refer to the project page or previous articles to check out the tasks and how xcom_push worked, but basically, it’s all the same.

If we push something, we can pull it also. We pull data from XCom in the following way:

```
def run_best_model(ti: Any, tag: str) -> str:

    """
    Runs the model with the best parameters searched and found
    at the earlier phase. Then, saves the model and info in the artifacts
    folder or bucket.
    """

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

    best_params = ti.xcom_pull(key="best_params", task_ids=["find_best_params"])
    best_params = best_params[0]
```

We pull the data from XCom with the xcom_pull method specifying the key we saved the data with (best_params) and the id of the task from which we pushed it (find_best_params). It returns a list, so we pick the first element. If we don't specify the key, xcom_pull defaults to return_value as the key, meaning we can write the code like this[9]:

```
best_params = ti.xcom_pull(task_ids=["find_best_params"])
best_params = best_params[0]
```

One caveat about XComs is that they can have any serializable value and are specially designed for small-size data such as task metadata, dates, model accuracy, or results. *Airflow* does not recommend that users pass large-size data such as dataframes. For this reason, when we work on our data, we don't transfer it between the tasks but save it and bring it from the local disk as follows:

```
from numpy import loadtxt


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

# 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=",")
```

We can retrieve data from the cloud and the local disk as well. It works well since we don't exchange the data between task instances and pass it through XCom.

Another recommendation by *Airflow* is to avoid using top-level imports and instead convert them to local ones inside *Python* callables. Top-level imports might take a lot of time and can generate a lot of overhead[10]. That's why we tried to write the code as seen in the upper block of the task:

```
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

    # Retrieve variables
    bucket, file_name, local_path, local_data_transformed_filename, _, _, _ = get_vars()
    original_file_name = file_name.split("/")[-1]

    # Read data
    data_transformed = pd.read_csv(local_data_transformed_filename)
```

Here, we import several libraries in the local namespace. However, I must confess that not always did I do local imports as recommended.

That's all with XComs. Now, it's time to focus on Variables.

**Variables**

*Variables are Airflow's runtime configuration concept — a general key/value store that is global and can be queried from your tasks, and easily set via Airflow's user interface, or bulk-uploaded as a JSON file.*

*XComs are a relative of Variables, with the main difference being that XComs are per-task-instance and designed for communication within a DAG run, while Variables are global and designed for overall configuration and value sharing[9].*

*Variables are global, and should only be used for overall configuration that covers the entire installation; to pass data from one Task/Operator to another, you should use XComs instead…Variables are really only for values that are truly runtime-dependent[11].*

In addition to this information that explains when to use XComs and Variables, another question may be why or when to use *AWS Parameter Store* if we have XComs and Variables. I don't have a clear-cut answer to this. I saved relatively stable variables on *Parameter Store* and *Airflow* tasks-related variables on *Airflow*. Yet, such an approach is still arguable and short of a perfect implementation.

We create a variable on *Airflow* with the set method:

```
from airflow.models import Variable


# Serialize and save/update mlflow_dict in Airflow variables.
mlflow_json_object = json.dumps(mlflow_dict, indent=2)
Variable.set(
    "mlflow_dict",
    mlflow_json_object,
```

```
    description="Mlflow variable names for 'experiment' and 'model'",
)
```

Given the information above, to save our *Python* dictionary mlflow_dict , which we introduced before, in *Airflow* Variables, we need to serialize it to JSON format and then upload it with the set method. We also upload the other two dictionaries (s3_dict and local_dict) similarly. We can find the code for this here:

**Machine-Learning/var_init.py at main · hsaltan/Machine-Learning**

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

github.com

Recall that we first introduced these three dictionaries from the define_variables task inside the dag file. Variables are stored as shown in Exhibits 15 and 16.

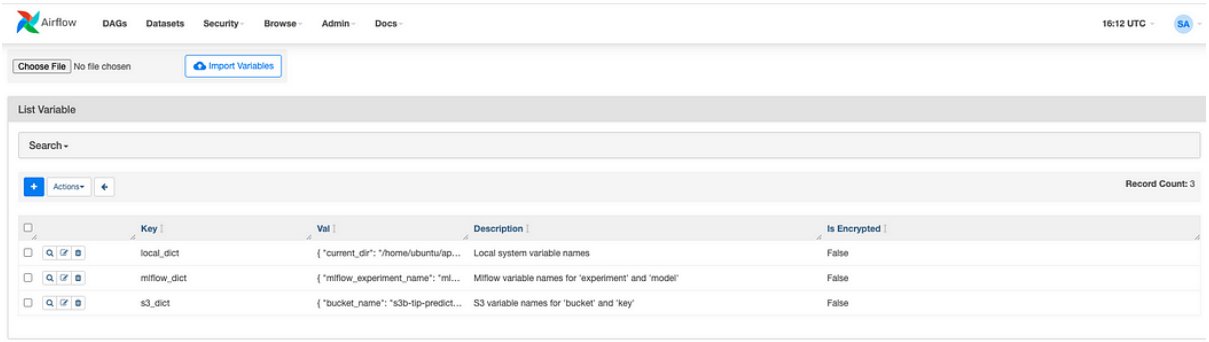



Exhibit-15: Airflow Variables List (Image by Author)

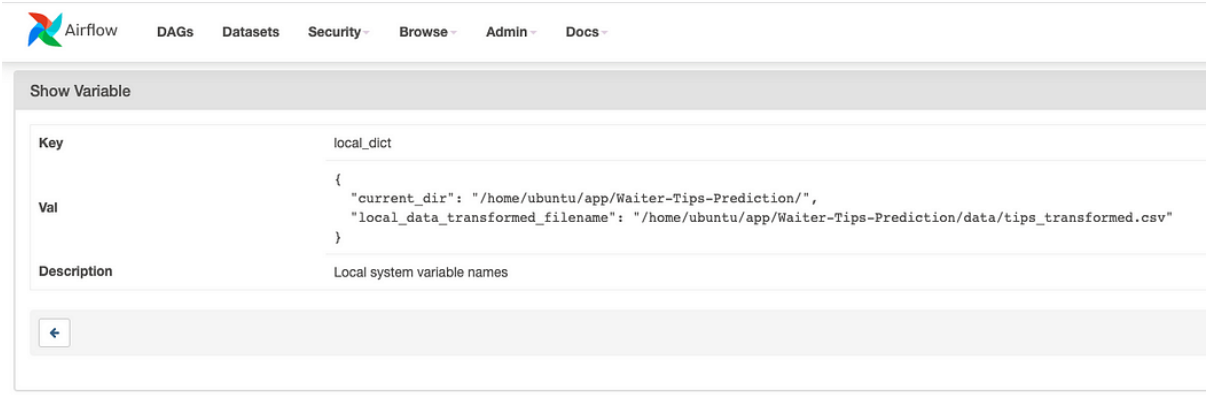



Exhibit-16: Airflow Variable local_dict (Image by Author)

Clicking “Admin” and “Variables” on the *Airflow* UI top menu, we can reach the Variables store.

To use variables, we call get on the Variable model:

```
# Retrieve the mlflow_dict variable
mlflow_dict = Variable.get("mlflow_dict", deserialize_json=True)
```

```
mlflow_experiment_name = mlflow_dict["mlflow_experiment_name"]
evidently_experiment_name = mlflow_dict["evidently_experiment_name"]
model_name = mlflow_dict["model_name"]
logging.info("Airflow variable dictionary stored as'%s' is retrieved.", mlflow_dict)
```

We deserialize the variable and convert it back to the *Python* dictionary. We do the same for the other two dictionaries. After setting variables once, we have had to get them in the scripts several times. Thus, it makes sense to organize the get variable operations by a utility script that we can find here:

**Machine-Learning/airflow_utils.py at main · hsaltan/Machine-Learning**

This is about machine learning projects. Contribute to hsaltan/Machine-Learning development by creating an account on...

github.com

Any time we need a variable, we retrieve it:

```
from utils.airflow_utils import get_vars


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

That is the *Airflow* part of the project so far. If you suffer enough to learn and do everything in *Airflow*, you can hope for the forgiveness of half of your sins! Testing for *Airflow* is also not so straightforward. We will see it in the following article while discussing several tools to test our code.

Before closing, I would like to talk about the issues I faced during the installation and execution of *Airflow* and how I could solve them.

**Troubleshooting**

1. Always check if the server and scheduler work normally by executing the following commands after you start them:

```
lsof -i tcp:8080
lsof -i tcp:8793
```

The first LiSt Open Files (lsof) command finds processes running on port 8080 (server), and the second command finds processes running on port 8793 (scheduler).

The output for the scheduler, for example, looks like this:

```
ubuntu@ip-10-0-1-68:~/app/Waiter-Tips-Prediction$ lsof -i tcp:8793
COMMAND    PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
gunicorn: 1397 ubuntu   11u  IPv6  30973      0t0  TCP *:8793 (LISTEN)
gunicorn: 1398 ubuntu   11u  IPv6  30973      0t0  TCP *:8793 (LISTEN)
gunicorn: 1399 ubuntu   11u  IPv6  30973      0t0  TCP *:8793 (LISTEN)
ubuntu@ip-10-0-1-68:~/app/Waiter-Tips-Prediction$
```

Exhibit-17: Checking Airflow Scheduler (Image by Author)

If you don't get any output for either or both of them, start the two processes over.

2. If you need to terminate the process of the server or scheduler, you should first run the lsof command to identify their process IDs (PID). For example, the first PID for the scheduler is 1397, as seen in Exhibit 17. Then, we can terminate the scheduler process by executing the following command:

kill 1397

3. Do not use if __name__ == '__main__': in dag files.

4. If some dags don't appear on Web UI, then something in the code must conflict with the *Airflow* logic. Another reason is that the scheduler doesn't run.

5. DAG ids must be unique.

6. Any custom module should reside in a folder with an empty __init__.py file, and the folder should be in the dags directory.

7. The path to any other file in the project folder should be defined in the dag file by a full name such as /home/ubuntu/app/Waiter-Tips-Prediction/data. Don't use relative paths like ../../web-flask/

8. When you run the *Airflow* web server, it will create the file $AIRFLOW_HOME/airflow-webserver-monitor.pid. If you try to re-run the *Airflow* web server daemon process, this will almost certainly produce the file $AIRFLOW_HOME/airflow-webserver.err. Execute the command below to fix the issue and restart the server:

rm $AIRFLOW_HOME airflow-webserver.err  airflow-webserver-monitor.pid
airflow webserver -p 8080 -D

Make sure *Airflow* must have created these files in the directory before deleting them.

9. When you run your *Airflow* scheduler, it will create the file $AIRFLOW_HOME/airflow-scheduler.pid. If you try to re-run the *Airflow* scheduler daemon process, this will almost certainly produce the file $AIRFLOW_HOME/airflow-scheduler.err. Execute the command below to fix the issue and restart the scheduler[12]:

rm $AIRFLOW_HOME airflow-scheduler.err  airflow-scheduler.pid
airflow scheduler -D

Make sure *Airflow* must have created these files in the directory before deleting them.

10. If DAGs don't appear on UI, or you receive the error "The scheduler does not appear to be running. The DAGs list may not update, and new tasks will not be scheduled.",

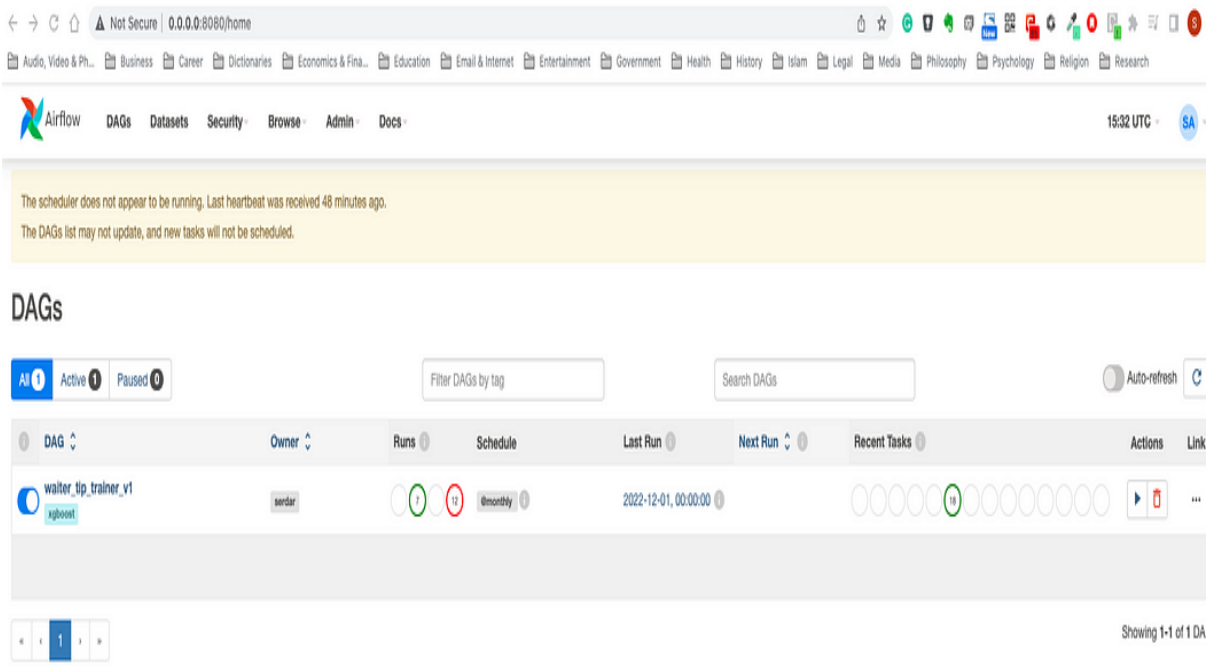



Exhibit-18: Errors Related to Airflow Scheduler (Image by Author)

restart the scheduler after removing the files mentioned in 9 above. Refresh the page.

11. *Airflow* may not recognize some typings for output (e.g., -> tuple[str, str], pd.DataFrame, list[Any], dict[str, str]) in the code on EC2. So, we may need to remove those typings. Instead, we can mention the return type in docstrings.

12. When I received the following error:

Broken DAG: [/home/ubuntu/app/Waiter-Tips-Prediction/dags/airflow_dag.py] Traceback (most recent call last):
  File "/usr/lib/python3.8/json/decoder.py", line 337, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
  File "/usr/lib/python3.8/json/decoder.py", line 353, in raw_decode
    obj, end = self.scan_once(s, idx)
json.decoder.JSONDecodeError: Expecting property name enclosed in double quotes: line 1 column 2 (char 1)

Broken DAG File "/usr/lib/python3.8/json/decoder.py", line 337, in decode
json.decoder.JSONDecodeError

I uninstalled the earlier version of *Airflow* (2.3.3), installed the newer version (2.4.2), and then restarted the airflow.db. That has solved the problem. Here, you should note that the *Airflow* version should be compatible with the *Python* version.

That is all with *Airflow*!

**Part 5: Unit Testing with Pytest and Apache Airflow Testing**

In the first three parts, we wrote the code for the core process, and in the last part, we designed it as a workflow that will run on a schedule. However, we cannot make sure if it works properly, if there is any bug, or if any runtime error occurs. We can remove the suspicion by implementing various testing tools in our code.

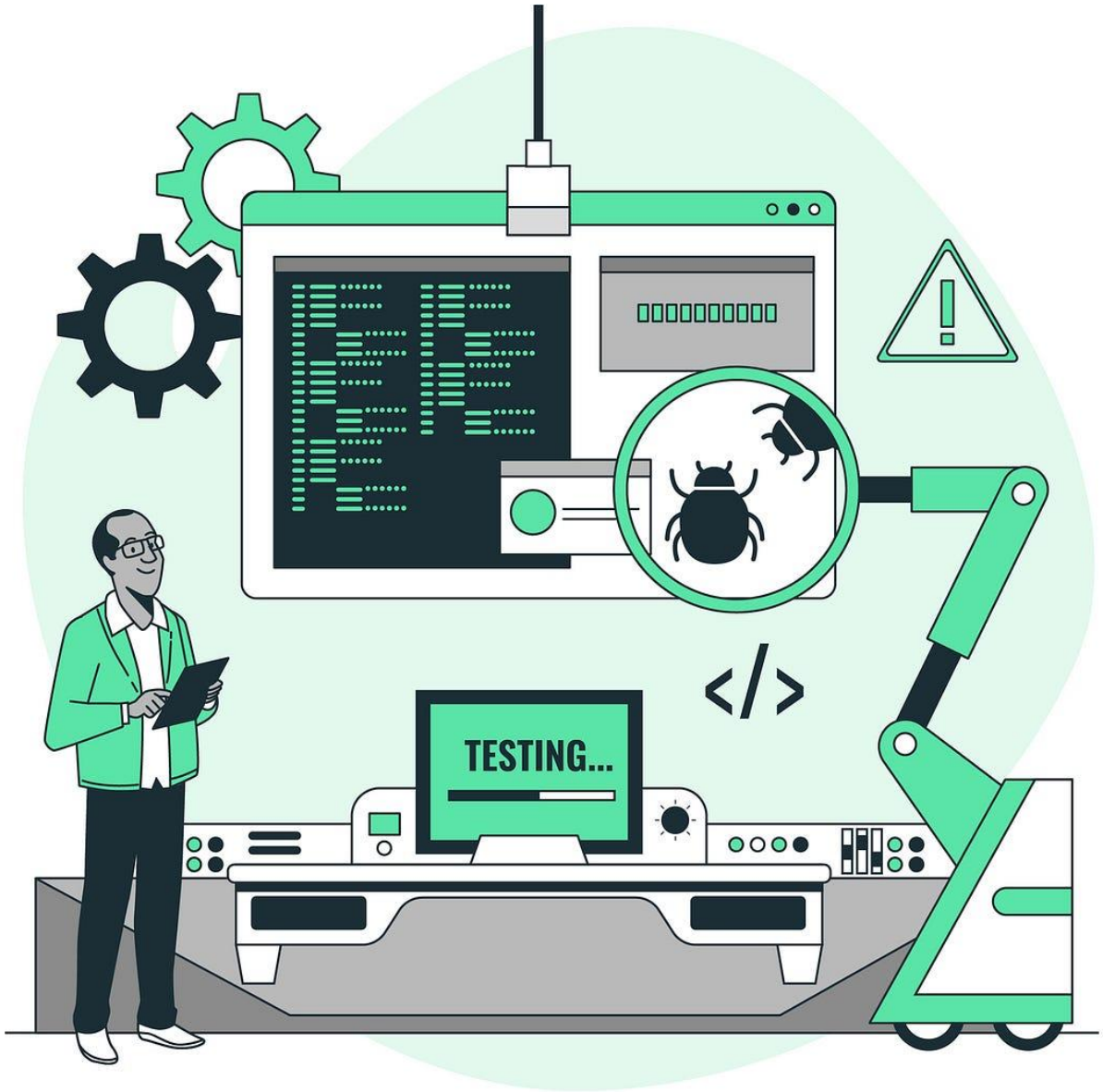



Image by storyset on Freepik

It may sound weird to suspect a malfunction since *Airflow* has run all functions successfully. Thus, we may think we have done the integration test.

*Integration testing exercises two or more parts of an application at once, including the interactions between the parts, to determine if they function as intended. This type of testing identifies defects in the interfaces between disparate parts of a codebase as they invoke each other and pass data between themselves.[1]*

*Airflow* ran all functions from all modules together from the start of the workflow to the end successfully. Therefore, we can mostly accept that the integration test is done already. Perhaps, we need to add the testing of XComs to complete the integration testing.

After the integration test is complete, it doesn't seem usual to do the unit testing, but this is what we will do! Otherwise, reading the articles may not get us to understand the project well. Whenever we finish writing a function, we should create the test function immediately to see if it works correctly. Suppose we type the following:

```
def sum_values(a, b):

  c = a + b
  return c
```

Then, we should write the following test function right after the sum_values to see if its execution is flawless:

```
def test_sum_values():

  result = sum_values(5, 12)
  assert result == 17
```

The test function should be a shadow function and allow us to check the health of the main one immediately. However, I didn't do it this way in this project. I made a mistake writing the test procedures after finishing the entire code.

We don't know what code errors and deficiencies we will face during tests. When they become clear, a technical burden will arise that will be time-consuming and tedious to eliminate. For such a small project, I was reluctant to do so. But, in more complex projects, this technical burden might be an Augean task. Therefore, we should develop and test the code simultaneously from the very beginning.

To perform testing, we'll use several tools. Let us start with *Pytest*.

Pytest

*Pytest* is a full-featured *Python* testing tool that helps check if our programs perform as they should. It performs unit testing and focuses on individual units of the source code, which are functions in our case. *Pytest* provides detailed information on the failing assert statements[2].

Installation

Install the latest version of *Pytest*:

```
pip install -U pytest
```

Check if it is installed:

```
pytest --version
```

Test Discovery and Execution

We have a few options for test discovery. The first is to use testpaths in pyproject.toml. PEP 518 states that pyproject.toml is a configuration file to store build system requirements for *Python* projects.

```
[tool.pytest.ini_options]
testpaths = "tests"
```

The testpaths variable “sets list of directories that should be searched for tests when no specific directories, files, or test ids are given in the command line when executing *pytest* from the rootdir directory.”[3] The “tests” phrase above is the name of the directory that contains our test folders and scripts.

Alternatively, from the definition of testpaths, it follows that we can also run pytest in the command line from the root directory (/home/ubuntu/app/Waiter-Tips-Prediction) as the second option.

```
pytest -v tests/
```

In the above command line, -v indicates verbose. The -v flag provides information such as test session progress, assertion details when tests fail, fixtures details, etc.

“ pytest determines a rootdir for each test run which depends on the command line arguments (specified test files, paths) and on the existence of configuration files.[4]”

Thus, if you open the terminal in the directory dir1/ , and specify the whole path to the test folder, such as /dir1/dir2/dir3/test_folder/, in the command line to run pytest, then pytest may consider dir1/ as the rootdir. As you may open the terminal in various directories for different runs, the rootdir will also change.

However, it’s safest to open the terminal in the directory where the configuration file (pyproject.toml) resides, such as /home/ubuntu/app/Waiter-Tips-Prediction , making it the rootdir for pytest.

If we don’t configure testpaths and don’t specify any directory to be tested in the command line, pytest will search for the current directory where the terminal is opened:

```
pytest
```

Opening the bash terminal in the project root directory, we can run pytest on a specific module for more granular testing,

```
pytest test_operators.py
```

or on a specific function (test_define_variables) in that module:

```
pytest test_operators.py::test_define_variables
```

The convention is that all sub-folders within the “tests” directory should start with test_. In these folders, *Pytest* searches for test_*.py or *_test.py files. And in these files, *Pytest* collects test-prefixed test functions or methods outside the class, and test-prefixed test functions or methods inside Test-prefixed test classes (without an __init__ method)[5]. By default, *Pytest* will consider any function prefixed with test as a test function, and any class prefixed with Test as a test collection.

Our test folder hierarchy will be as in Exhibit 1:

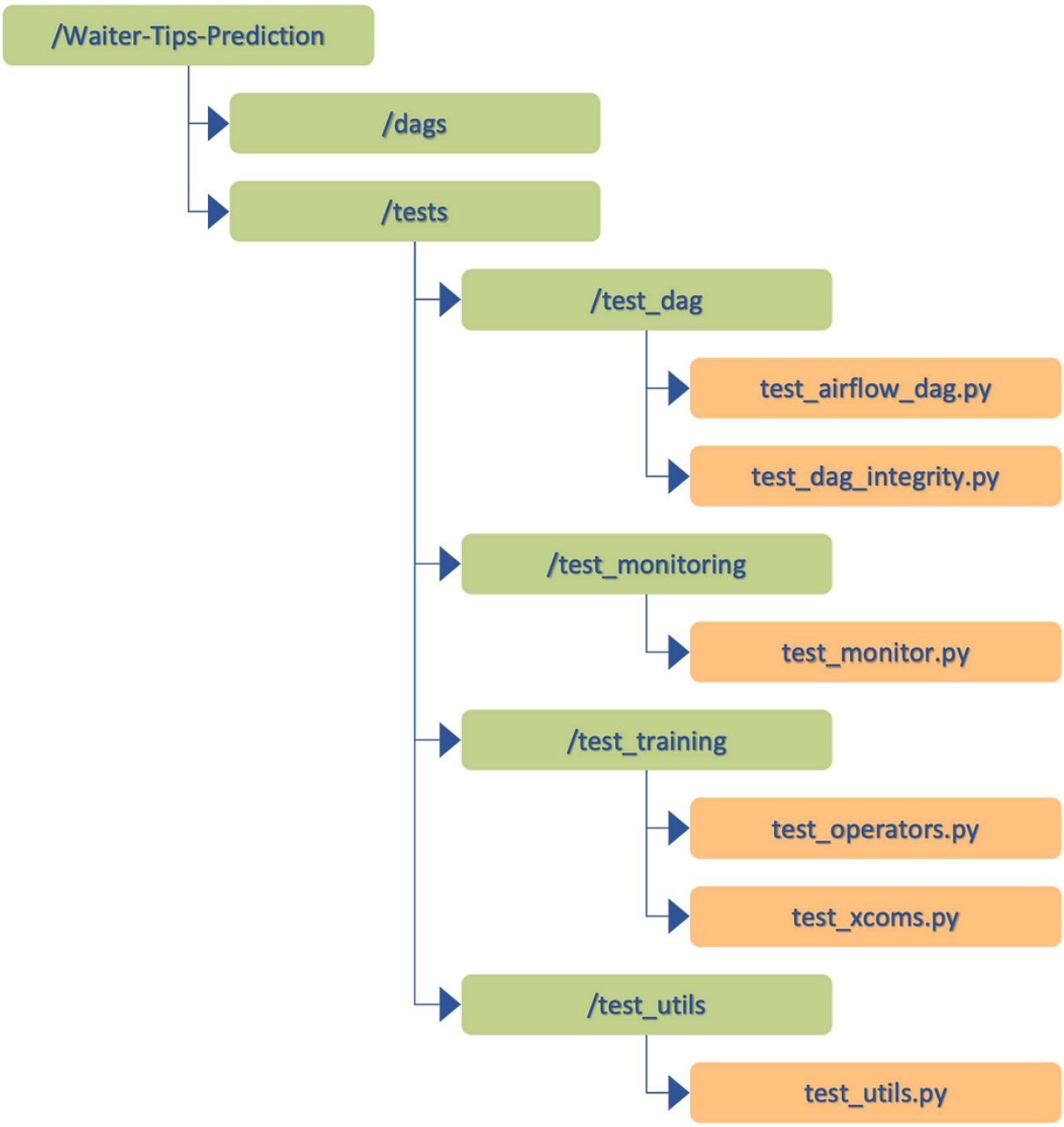



Exhibit-1: Tests Folder Hierarchy (Image by Author)

Using test as a prefix is, of course, not a command from the Word of God! Any name is good to go as long as *Pytest* knows the practice applied[6]. For example, we can even use "hakunamatata" as the prefix!

Image by brgfx on Freepik

We make *Pytest* aware of “hakunamatata” thanks to the pyproject.toml file:

```
[tool.pytest.ini_options]
python_files = hakunamatata_*.py
python_classes = Hakunamatata
python_functions = *_hakunamatata
```

*Pytest* would look for tests in files that match the hakunamatata_*.py glob-pattern, Hakunamatata prefixes in classes, and functions and methods that match *_hakunamatata. For example, a test function will have the following format:

```
def func_hakunamatata(self):
   pass
```

We can check for multiple glob patterns by adding a space between them.

The third option to run *Pytest* is through VSC. We open the Command Palette by clicking Ctrl+Shift+P (Cmd+Shift+P on Mac), type “test,” and choose in order “Python: Configure Tests,” “pytest” and “tests.” The “tests” folder is user-defined.

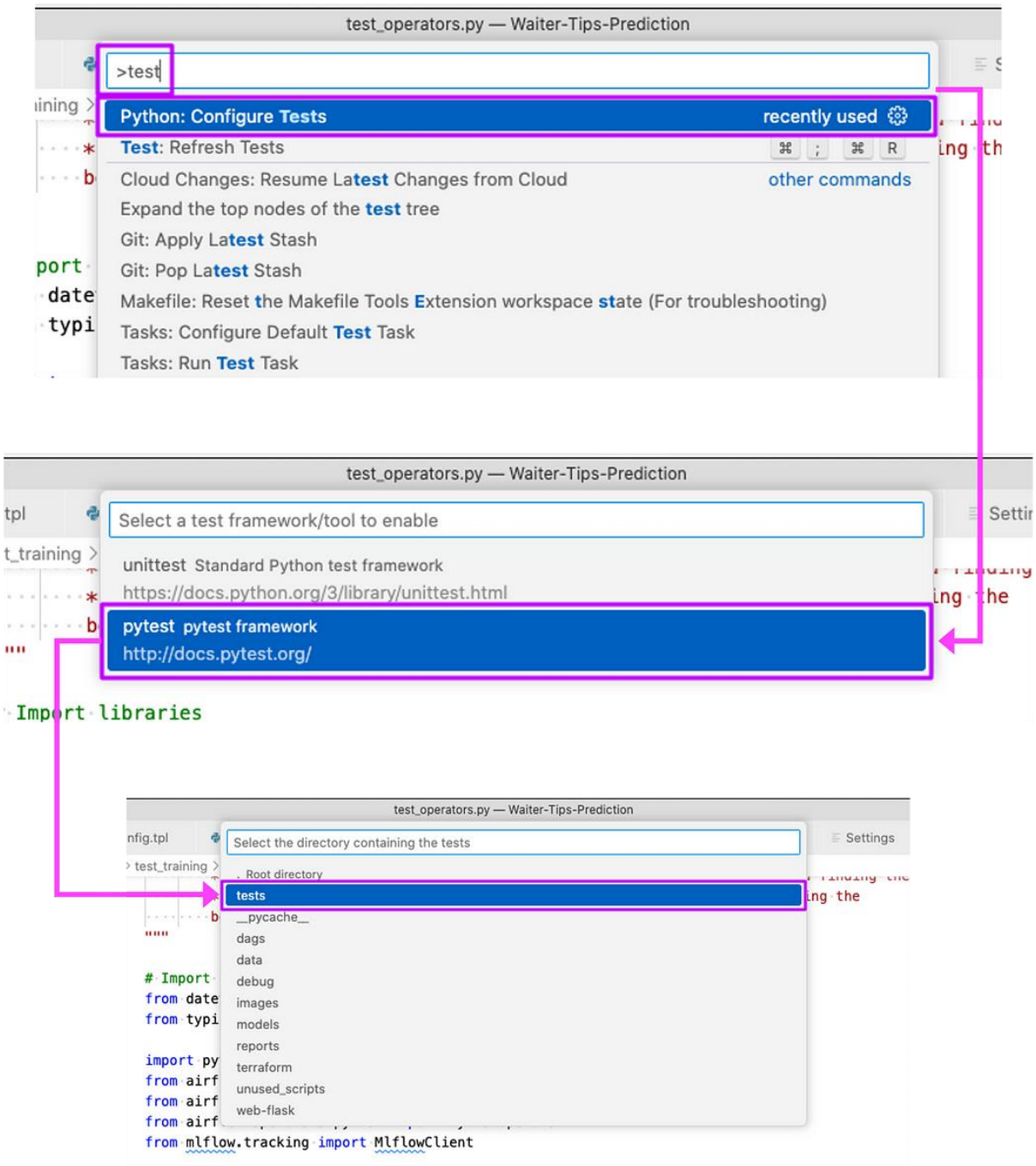



Exhibit-2: Configuring Pytest on VSC (Image by Author)

VSC will discover *Pytest* in the "Test Explorer," represented by the test beaker icon on the left. We can trigger test discovery at any time using the "Test: Refresh Tests" command from the Command Palette.

When we click the Run Tests button, testing will start.

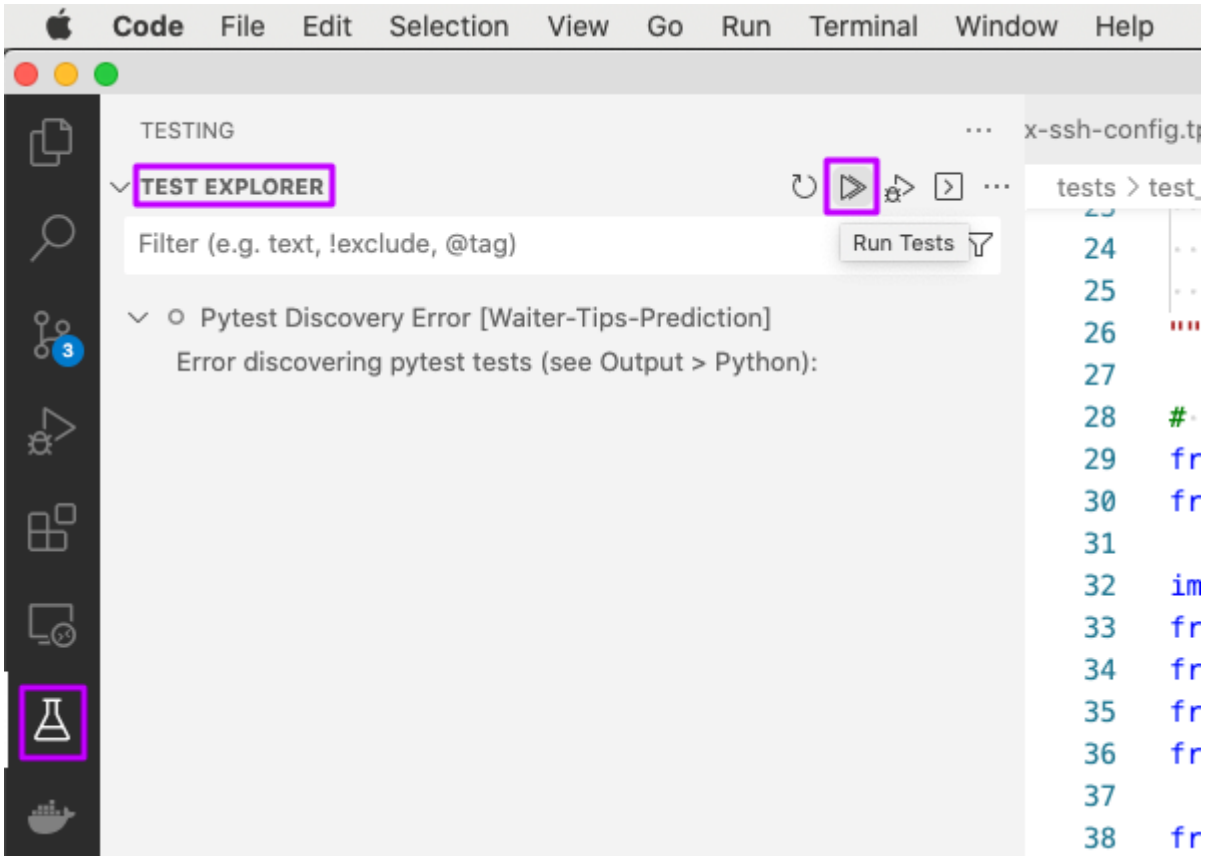



Exhibit-3: Run Pytest on VSC (Image by Author)

We can also run the tests from the Command Palette:

*Test: Run All Tests — Runs all tests that have been discovered.*

*Test: Run Tests in Current File — Runs all tests in a file that is open in the editor.*

*Test: Run Test at Cursor — Runs only the test method under your cursor in the editor.[7]*

We might encounter a discovery issue. We'll address it in Troubleshooting later.

In this project, it became necessary to add an empty __init__.py into the tests folder and subfolders, thus changing them to packages (we didn't show __init__.py files in Exhibit 1 to avoid crowding.) Sometimes, VSC cannot discover tests placed in subfolders because such test files cannot be imported. To make them importable, we should create __init__.py in that folder[7].

*Skipping Files*

There are a few ways to skip a file or function in *Pytest*. We can summarize them[8]:

1. Mark the function with the skip decorator to skip it, such as:

```
@pytest.mark.skip(reason="we don't wanna test this for now")
def test_split_data(the_dag: Any) -> None:
  ...
```

2. Skip all tests in a module unconditionally:

```
pytestmark = pytest.mark.skip("all tests not ready yet")
```

3. Skip all tests in a module based on some condition:

pytestmark = pytest.mark.skipif(sys.platform == "win32", reason="tests for linux only")

The last two are written at the module level. You may refer to the documentation to get more information about skip.

Skipping files using the command line is through passing the --ignore=path option. *Pytest* allows multiple uses of this option in the same line.

pytest --ignore=tests/test_monitoring/test_monitor.py --
ignore=tests/test_training/test_operators.py

Pytest will collect all test modules except test_monitor.py and test_operators.py modules[6].

Layout

*Pytest* supports two typical test layouts: “Tests outside application code” and “Tests as part of application code.” Our layout seems to be of the first type as the “tests” directory is outside the application folder, which is “dags” (see Exhibit 1). *Pytest* explains and shows the first type as:

*Putting tests into an extra directory outside your actual application code might be useful if you have many functional tests or for other reasons want to keep tests separate from actual application code (often a good idea)[5]:*

```
pyproject.toml
src/
   mypkg/
      __init__.py
      app.py
      view.py
tests/
   test_app.py
   test_view.py
   ...
```

In the case of “tests as part of application code,” our “tests” directory would have been placed under the “dags” folder. This way is valuable if a direct relation between tests and application modules exists and we want to distribute tests along with the application[5]. Then, it would have been possible to run the test by executing:

pytest --pyargs dags

*Pytest* would have then discovered where “dags” was installed and collected tests from there.

As Exhibit 1 suggests, we don’t have a direct relation between tests and application modules, and tests are already outside the application code. It was just mentioned here as an option we could have chosen.

We have seen how to discover and execute tests and can go on to the generation of tests.

Writing Tests

We know how to discover and run the tests but don't have any of them written yet! So, we should start with creating them. We are going to do this in two sections. The first is testing for the regular code, and the second is one for the *Airflow* code.

*Tests for the Regular Code*

This section covers testing all code that doesn't contain *Airflow* elements. Showing a few examples here should explain the subject. Let us start with the test for the utils' functions. Utils modules can be found here.

- Testing put_object method uploading a file in the S3 bucket:

```
# Import libraries
import boto3
from dags.utils.aws_utils import put_object


def test_put_object() -> None:

    """
    Tests putting an existing object in the S3 bucket.
    """

    bucket = "s3b-tip-predictor"
    key = "data/tips.csv"

    # Upload a dummy object into the S3 bucket
    put_object(
        "/home/ubuntu/app/Waiter-Tips-Prediction/data/tips.csv",
        bucket,
        key,
        "Name",
        "data",
    )

    # Get the tag of the uploaded object to verify.
    # It is correct if the object is uploaded with the given tag successfully
    response = boto3.client("s3", region_name="eu-west-1").get_object_tagging(
        Bucket=bucket, Key=key
    )

    tag_key = response["TagSet"][0]["Key"]
    tag_value = response["TagSet"][0]["Value"]

    assert tag_key == "Name"
    assert tag_value == "data"
```

We first upload an object in S3 by executing put_object function, which we defined in the utils module. Then, we retrieve the tag key and value by running the *AWS*

*Boto3* library's get_object_tagging method. We compare tag elements returned by the user-defined function with those obtained from the *Boto3* implementation.

- Testing our user-defined list_experiments function, which gets the list of experiments in *MLflow*:

```
# Import libraries
from mlflow.tracking import MlflowClient
from dags.utils.mlflow_utils import list_experiments


mlflow_client = MlflowClient("http://127.0.0.1:5000")


def test_list_experiments() -> None:

    """
    Tests getting the list of experiments.
    """

    # List of all experiments with full information
    experiments = list_experiments(mlflow_client)

    # List of experiments by name only
    experiment_names = [experiment.name for experiment in experiments]

    assert "mlflow-experiment-1" in experiment_names
```

When we set the experiment, we named it "mlflow-experiment-1." With the list_experiments , we check if it exists among all *MLflow* experiments.

Now, let us see an example from the monitoring module.

- Testing *Evidently* dashboards:

```
# Import libraries
from dags.monitoring.evidently_monitoring import monitor_evidently


def test_monitor_evidently() -> None:

    """
    Tests recording data drift evaluation results in MLFlow.
    """

    metrics = monitor_evidently(1)

    assert (
        str(metrics["evidently_1"])
        == "gauge:evidently:num_target_drift:count:reference"
```

```
    )
    assert (
        str(metrics["evidently_68"])
        == "gauge:evidently:regression_performance:feature_error_bias:total_bill_num_ref_over"
    )
    assert (
        str(metrics["evidently_195"])
        ==
"gauge:evidently:data_quality:quality_stat:reference_total_bill_num_most_common_value"
    )
    assert (
        str(metrics["evidently_278"])
        == "gauge:evidently:data_quality:quality_stat:current_size_num_mean"
    )
```

The monitor_evidently function from the evidently_monitoring module computes data metrics and translates them to *Prometheus* for querying and displaying them on *Prometheus* and *Grafana*. We check here if that is done correctly. monitor_evidently takes a keyword argument: flag=0, which means it will not return anything if the value is zero. So, we input its value as 1 in the test function. *Prometheus*-translated metrics were stored in a dictionary as:

```
metrics[f"evidently_{i}"] = prom_metric
```

We randomly chose the metrics, such as evidently_68 or evidently_195, and tried to see if their values matched those in the metrics dictionary. That was one of the tests that we could do for the monitoring module.

*Tests for the Airflow Code*

Inspecting the code written for *Airflow* requires special treatment. We will test the DAG, variables, operators, and XComs. For this, we first need to know what “fixtures” are.

*Fixtures*

Fixtures provide a fixed baseline that each test function (related) it is applied to will repeat. They are used to provide data that the “related test functions” will use as input, thereby removing the unnecessary work of repeating the same code.

We introduce a fixture at the top of the “fixture function” as @pytest.fixture. “Related test functions” access the data made available by fixtures through arguments. For each fixture used by a test function, a parameter with the same name as the “fixture function” is put in the test function’s definition.

The explanation will become precise when we see the examples.

*Testing the DAG*

We start with testing the dag file.

```
# Import libraries
from typing import Any
```

```
import pytest
from airflow.models import DagBag


@pytest.fixture
def dag_bag() -> Any:

    """
    Sets the dagbag as a fixture for repetitive use.
    """

    dagbag = DagBag(include_examples=False)

    return dagbag

@pytest.fixture
def the_dag(dag_bag: Any) -> Any:

    """
    Sets the dag as a fixture for repetitive use.
    """

    dag_id = "waiter_tip_trainer_v1"
    dag = dag_bag.get_dag(dag_id)

    return dag
```

Here, we use a chain of fixtures. We first import the DagBag library from *Airflow*. A dagbag is a collection of dags. The first function returns the existing DAGs in the collection, excluding the DAG examples. The second function uses the returned value (existing DAGs) as input and gets the DAG, which has the ID “waiter_tip_trainer_v1” from among returned DAGs.

We had to use DagBag to get our DAG as get_dag is a method of classairflow.models.dagbag.DagBag class in *Airflow*[9]. Notice that the_dag uses dag_bag as a parameter that is the same as the name of the higher function (dag_bag). The test functions that need our DAG will use the_dag as their argument. Similarly, any test function that needs to get the collection of dags will use dag_bag as its parameter.

- Testing if any import errors exist:

```
def test_dagbag(dag_bag: Any) -> None:

    """
    Tests for dag integrity.
    """

    assert not dag_bag.import_errors
```

- Testing if the DAG has a tag (not empty):

```
def test_tag(the_dag: Any) -> None:

    """
    Tests the dag if any tags exist.
    """

    assert the_dag.tags
```

- Checking the number of tasks in the DAG:

```
# Dag definition
def test_task_count(the_dag: Any) -> None:

    """
    Test for dag definition, checking the number of tasks
    in the dag.
    """

    assert len(the_dag.tasks) == 18
```

- Checking if the DAG has all tasks:

```
def test_contain_tasks(the_dag: Any) -> None:

    """
    Tests the dag for checking the tasks it contains.
    """

    tasks = the_dag.tasks
    task_ids = list(map(lambda task: task.task_id, tasks))
    assert sorted(task_ids) == sorted(
      [
        "start_dag",
        "define_variables",
        "initialize_vars",
        "transform_data",
        "split_data",
        "search_best_parameters",
        "find_best_params",
        "run_best_model",
        "register_best_model",
        "test_model",
        "compare_models",
        "get_top_run",
        "prepare_evidently_data",
        "create_evidently_reports",
        "evaluate_data_drift",
        "record_metrics",
        "monitor_evidently",
        "end_dag",
```

```
    ]
)
```

These tasks are in the dag file.

- Checking for dependencies between tasks:

```
def test_dependencies(the_dag: Any) -> None:

    """
    Tests the dag for checking dependencies between tasks.
    """

    task_mlflow_search = the_dag.get_task("search_best_parameters")
    task_split_data = the_dag.get_task("split_data")
    task_create_evidently_reports = the_dag.get_task("create_evidently_reports")
    task_prepare_evidently_data = the_dag.get_task("prepare_evidently_data")

    upstream_task_ids_1 = list(
        map(lambda task: task.task_id, task_mlflow_search.upstream_list)
    )
    assert ["split_data"] == upstream_task_ids_1

    upstream_task_ids_2 = list(
        map(lambda task: task.task_id, task_split_data.upstream_list)
    )
    assert sorted(["define_variables", "transform_data"]) == sorted(upstream_task_ids_2)

    downstream_task_ids_1 = list(
        map(lambda task: task.task_id, task_create_evidently_reports.downstream_list)
    )
    assert not downstream_task_ids_1

    downstream_task_ids_2 = list(
        map(lambda task: task.task_id, task_prepare_evidently_data.downstream_list)
    )
    assert sorted(["create_evidently_reports", "evaluate_data_drift"]) == sorted(
        downstream_task_ids_2
    )
```

We make sure tasks follow each other in the right order. Here, we control four tasks. For example, split_data should come
before search_best_parameters. create_evidently_reports and evaluate_data_drift should follow prepare_evidently_data.

- Testing DAG integrity:

Here, we need to know another feature of *Pytest*: test parametrization. Parameterization of a test allows us to run the test against multiple sets of inputs. For example[10]:

```
# content of test_expectation.py
import pytest

@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

The @parametrize decorator defines three different (test_input, expected) tuples as input. The test_eval function will compare the first element result (test_input, e.g "3+5") of a tuple in the list with the second element (expected, e.g. 8) of the tuple. It will run three times to show the test results. That is the same as:

```
assert eval("3+5") == 8
assert eval("2+4") == 6
assert eval("6*9") == 42
```

Now, we can return to our DAG[11]:

```
# Import libraries
import glob
from os import path
from types import ModuleType
from typing import Any

import pytest
from airflow import models as airflow_models
from airflow.utils.dag_cycle_tester import check_cycle


DAG_PATHS = glob.glob(path.join(path.dirname(__file__), "..", "..", "dags", "*.py"))

@pytest.mark.parametrize("dag_path", DAG_PATHS)
def test_dag_integrity(dag_path: Any) -> None:

    """
    Imports DAG files and checks for a valid DAG instance.
    """

    dag_name = path.basename(dag_path)
    module = _import_file(dag_name, dag_path)

    # Validate if there is at least 1 DAG object in the file
    dag_objects = [
        var for var in vars(module).values() if isinstance(var, airflow_models.DAG)
    ]
    assert dag_objects

    # For every DAG object, test for cycles
    for dag in dag_objects:
```

```
    check_cycle(dag)

def _import_file(module_name: str, module_path: str) -> ModuleType:

  """
  Loads the module.
  """

  import importlib.util

  spec = importlib.util.spec_from_file_location(module_name, str(module_path))
  module = importlib.util.module_from_spec(spec)
  spec.loader.exec_module(module)

  return module
```

We collect the basenames and file paths of every dag file in the dags folder. We currently have one dag file (airflow_dag.py). Then, we find and load the module of each dag file and check if at least one valid DAG object is in the file. We created the DAG object with the with context during DAG instantiation and set any DAG-level parameters, such as the schedule interval.

As we already know, DAG is supposed to be acyclic. With the check_cycle method, we check if any task cycles exist in the DAG. It returns False if no cycle is found, otherwise raises an exception. All tasks must follow a directed acyclic schema.

*Testing Airflow Variables*

We will find if variables are correctly stored and retrieved from *Airflow* Variables.

```
from dags.utils.airflow_utils import get_vars


def test_get_vars() -> None:

  """
  Tests all variables created and stored on
  Airflow Variables.
  """

  # Retrieve all variables stored in Airflow
  (
    bucket,
    file_name,
    local_path,
    local_data_transformed_filename,
    mlflow_experiment_name,
    evidently_experiment_name,
    model_name,
  ) = get_vars()
```

```
    assert bucket == "s3b-tip-predictor"
    assert file_name == "data/tips.csv"
    assert local_path == "/home/ubuntu/app/Waiter-Tips-Prediction/"
    assert (
        local_data_transformed_filename
        == "/home/ubuntu/app/Waiter-Tips-Prediction/data/tips_transformed.csv"
    )
    assert mlflow_experiment_name == "mlflow-experiment-1"
    assert evidently_experiment_name == "evidently-experiment-1"
    assert model_name == "xgboost-model"
```

The get_vars function defines and stores variables in *Airflow*. We added a return statement in get_vars so that it can deliver all these variables. The test function above compares the delivered results with the variable names we defined in get_vars to check if they match. If they do, the function has sent the variables to *Airflow and* retrieved them from there correctly.

*Testing Airflow Operators*

We used three operators: EmptyOperator, BashOperator, and PythonOperator. We'll run tests for the last two.

Let us define the fixture function for repetitive use:

```
# Import libraries
from datetime import datetime
from typing import Any

import pytest
from airflow.models import DAG, Variable


@pytest.fixture
def the_dag() -> Any:

    """
    Sets the dag as a fixture for repetitive use.
    """

    dag = DAG(dag_id="waiter_tip_trainer_v1", start_date=datetime(2022, 8, 25, 2))

    return dag
```

We are not going to see all operator tests but a few examples to understand the topic.

- Testing BashOperator:

```
from airflow.operators.bash import BashOperator


def test_bash_operator(the_dag: Any) -> None:
```

```
    """
    Tests the bash operator.
    """

    test = BashOperator(
        dag=the_dag, task_id="initialize_vars", bash_command="echo hello"
    )
    result = test.execute(context={})

    assert result == "hello"
```

We send a command (display “hello” on the screen) to the BashOperator — initialize_vars task in our DAG to execute it, then check if it works correctly.

When we run a function in *Airflow*, we may pass some keyword arguments to it. Here, context on the execute() method collects and passes those keyword arguments to the operator.
As BashOperator doesn’t require any variable to be passed, we left the context empty with {}.

- Testing PythonOperator:

```
from airflow.operators.python import PythonOperator
from dags.training.data_preprocessor import split_data


def test_split_data(the_dag: Any) -> None:

    """
    Tests splitting the data as training and validation sets, and
    saving them to the local disk and to the S3 bucket.
    """

    task = PythonOperator(dag=the_dag, task_id="split_data", python_callable=split_data)
    result = task.execute(context={"test_size": 0.25})

    assert result[0] == 183
    assert result[1] == 61
    assert result[0] + result[1] == 244
```

Here, we almost mimic the content of task_split_data in the dag file (airflow_dag.py). We ask the task split_data in the dag file to run the *Python* callable split_data of
the data_preprocessor module with the test_size argument, which has a value of 0.25. If you recall, the original test size in the dag file was 0.20. For a quick reminder, task_split_data was:

```
TEST_SIZE = 0.2

task_split_data = PythonOperator(
    task_id="split_data",
    python_callable=split_data,
    op_kwargs={"test_size": TEST_SIZE},
```

```
    do_xcom_push=False,
)
```

In the test function, the context argument passes the new test_size to see if split_data correctly divides the data with the updated split ratio.

Let us see one more operator example and finish the test for operators. Other tests for Python operators show a similar pattern already.

```
from dags.training.data_processor import search_best_parameters

def test_search_best_parameters(the_dag: Any) -> None:

    """
    Tests searching and finding the optimum parameters within the
    defined ranges with Hyperopt.
    """

    import math

    task = PythonOperator(
        dag=the_dag,
        task_id="search_best_parameters",
        python_callable=search_best_parameters,
    )
    result = task.execute(context={"tag": "test_xgboost"})

    # Float values may differ, so it makes sense to check equality with tolerance
    assert isinstance(result["learning_rate"], float) is True
    assert math.isclose(round(float(result["subsample"]), 2), 0.80, abs_tol=0.2) is True
```

This test checks if optimized hyperparameters, learning_rate, and subsample, found by the search_best_parameters function of the data_processor module are float values and if the subsample is within an acceptable interval.

Some of my functions became lengthier than they should be and eventually involved more variables than advised. I could have prevented it from happening and some other errors if I had started to test the units after writing them and periodically implemented tests over the course of the code development.

*Testing Airflow XComs*

Tests for XCom become a part of the integration test as XCom exchange data between tasks. We will use the two fixture functions written earlier: dag_bag and the_dag.

```
# Import libraries
import math

from airflow.models.taskinstance import TaskInstance
```

```
def test_xcoms_rmses(the_dag: Any) -> None:

    """
    Checks the XCom variable for the RMSEs.
    """

    # Get the push and pull tasks of the dag
    push_to_xcoms_task = the_dag.get_task("test_model")
    pull_from_xcoms_task = the_dag.get_task("compare_models")

    # Set the run_id for which to check the relevant xcom value.
    run_id = "manual__2022-09-20T08:52:33.473203+00:00"

    # Run the push task and push the variable to XCom
    push_to_xcoms_ti = TaskInstance(task=push_to_xcoms_task, run_id=run_id)
    context = push_to_xcoms_ti.get_template_context()
    push_to_xcoms_task.execute(context)

    # Run the pull task and pull the variable from XCom
    pull_from_xcoms_ti = TaskInstance(task=pull_from_xcoms_task, run_id=run_id)
    result = pull_from_xcoms_ti.xcom_pull(key="rmses")

    # Float values may differ, so it makes sense to check equality with tolerance
    assert (
        math.isclose(
            round(float(result["model_production_rmse"]), 4), 0.8100, abs_tol=0.2
        )
        is True
    )
    assert (
        math.isclose(round(float(result["model_staging_rmse"]), 4), 0.8100, abs_tol=0.2)
        is True
    )
```

In the data_processor.py module, the test_model task pushed the newly computed RMSE (model_staging_rmse) and the RMSE of the model (model_production_rmse), which is already in production, to XComs in the following way:

```
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="rmses", value=rmse_dict)
```

And the compare_models function pulled RMSEs from the XComs:

```
# Get rmse_dict from XCom
rmse_dict = ti.xcom_pull(key="rmses", 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"]
```

In the test_xcoms_rmses function above, we imitate these two operations. We designate test_model and compare_models as push and pull tasks, respectively. Introducing their ids, we get these tasks from our DAG and assign them to variables push_to_xcoms_task and pull_from_xcoms_task.

Choosing a run_id for testing purposes, we receive each task's instance that was created during a particular run represented by this run_id.

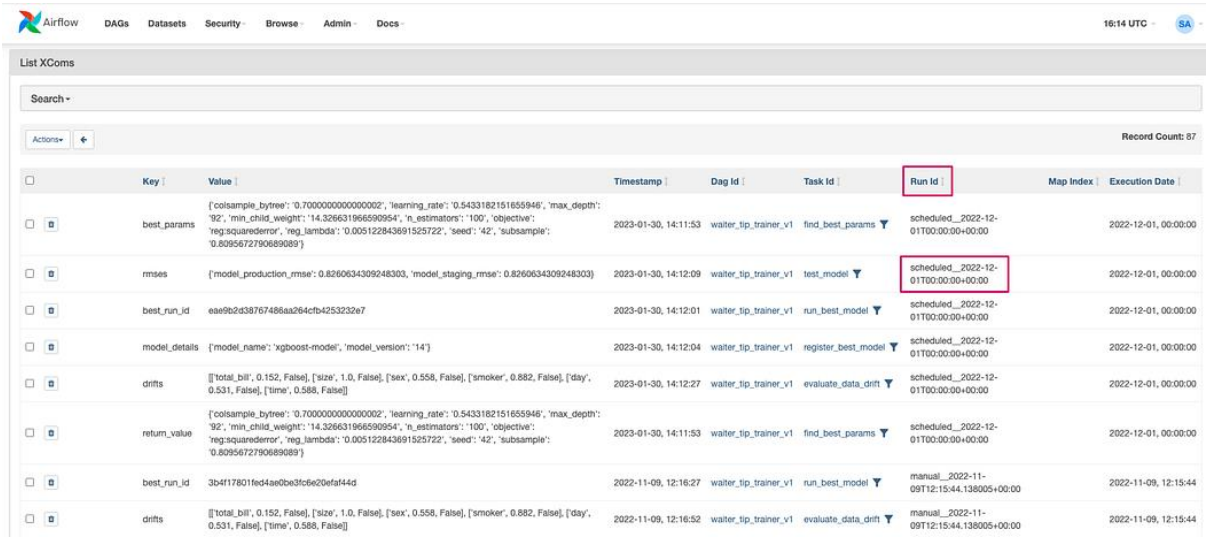



Exhibit-4: Run Id Information in Airflow Xcoms (Image by Author)

There are templates in *Airflow*. They allow us to pass dynamic data to our task instances at runtime. For example, we may need to get today's date, our run id, user-defined params, or access *Airflow* Variables during the runtime. Templates with variables, macros, and filters can help us get and pass such data[12]. We can generate the template context from the TaskInstance as well. Template context is a dictionary that is passed to the task.

In the above test code, we retrieve the template context from the task instance of test_model with the get_template_context method and assign it to the context variable, a dictionary. In our case, it is the rmse_dict dictionary we pushed to XComs with test_model. That's why get_template_context could help us generate the context from this task. Then, we execute the test_model task with the context argument (pushing rmse_dict to XComs), but this time for a test.

The compare_models task retrieves the dictionary from XComs using the key (key="rmses") as we did in the original code. Finally, we check the values of RMSEs within a defined tolerance margin. With the successful push and pull cycle, we learn that XComs works smoothly.

This way, we test all XComs push and pull operations in the test_xcoms.py module. One who wants to see all test files can find them here:

Machine-Learning/Waiter-Tips-Prediction/tests at main · hsaltan/Machine-Learning

[You can’t perform that action at this time. You signed in with another tab or window. You signed out in another tab or...](https://github.com)

github.com

We have completed *Pytest* here. In the Troubleshooting section, we will discuss how to fix some errors that might arise during the *Pytest* execution.

I planned to address testing in one article, but it became longer than expected. So, I’ve decided to distribute it across a few parts. That would unavoidably extend the series. In the following article, we will continue with other testing tools.

Troubleshooting

While configuring *Pytest* on VSC, the editor might not discover the test.

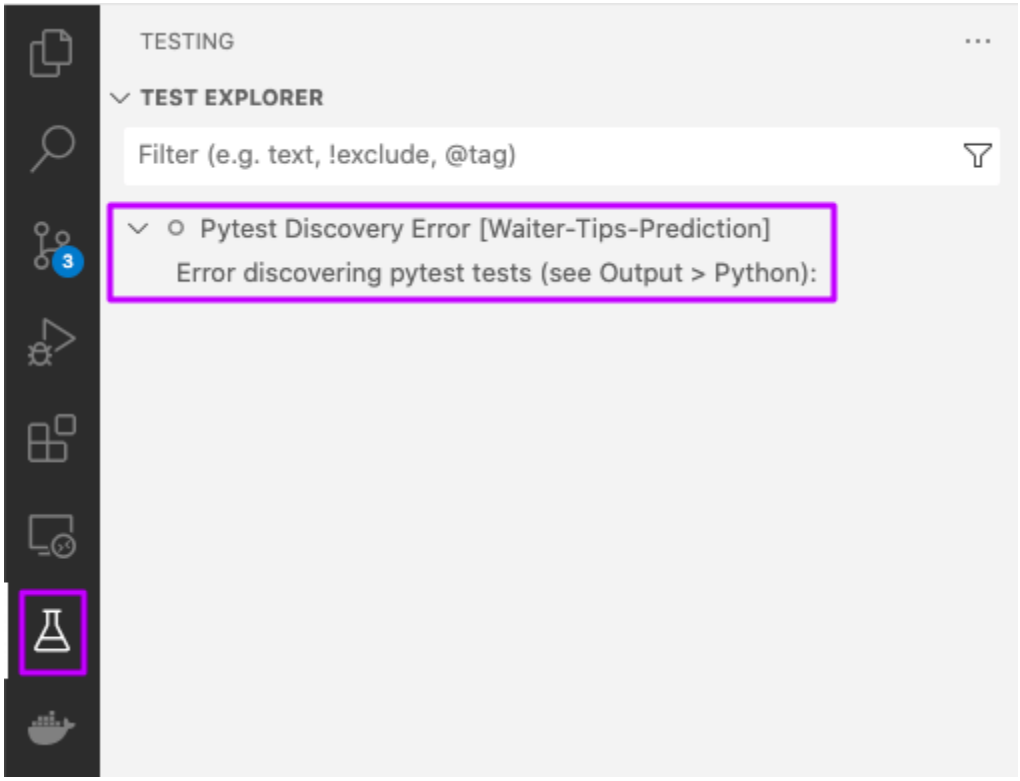



Exhibit-5: Pytest Discovery Error VSC (Image by Author)

Then, we need to look at the output.

Exhibit-6: Test Explorer Output (Image by Author)

We find the output on the “Output” tab of the VSC terminal. We should also select “Python” in the dropdown on the right. When we read the message, we see that *Pytest* collected 23 tests overall and faced a problem with the two. And the error was ModuleNotFoundError: No module named ‘utils’. This module is an internal library. In some cases, the “module not found” error might arise from an issue related to a third-party library. Then, the issue is straightforward: install the third-party library.

If the “module not found” error is due to an internal library like ours, we should pay attention to the following:

1. First, open the settings.json file in VSC and make sure that python.testing.autoTestDiscoverOnSaveEnabled argument exists and is set to True. It should be in the settings file automatically, but it might not be. python.testing.autoTestDiscoverOnSaveEnabled will ensure automatic discovery whenever we add, delete, or update any *Python* file in the workspace[7].

Second, check if python.testing.pytestEnabled exists and is set to True too. Therefore, you should have in settings.json file as with other parameters :

"python.testing.pytestEnabled": true,
"python.testing.autoTestDiscoverOnSaveEnabled": true

We will need to reload the window for this setting to take effect.

2. Next, we should avoid using relative paths for import; instead, use absolute paths. A relative path is like this[13]:

from .module2 import function1

An absolute path is:

from package1.module2 import function1

3. If converting the relative path to the absolute path doesn’t solve the problem, use the PYTHONPATH environment variable.

*Pythonpath is a special environment variable that provides guidance to the Python interpreter about where to find various libraries and applications…Python can find its standard library. So, the only reason to use PYTHONPATH variables is to maintain directories of custom Python libraries that are not installed in the site packages directory (the global default location). In simple terms, it is used by user-defined modules to set the path so that they can be directly imported into a Python program.[14]*

On Linux, we can set the PYTHONPATH variable by opening the bash profile,

nano /home/ubuntu/.bashrc

adding the PYTHONPATH variable,

export PYTHONPATH="/home/ubuntu/app/Waiter-Tips-Prediction"

or appending to the existing PYTHONPATH if it already exists,

export PYTHONPATH="${PYTHONPATH}:/home/ubuntu/app/Waiter-Tips-Prediction/"

saving and exiting the bash profile, and finally reloading the profile.

source .bashrc

One can refer to this page for how to set the PYTHONPATH variable on Mac and Windows.

The discovery of tests by VSC may sometimes be challenging to do. When I did everything written above, VSC still failed to discover the tests, and I kept getting the ModuleNotFoundError: No module named 'utils'. I resolved the issue using relative imports!

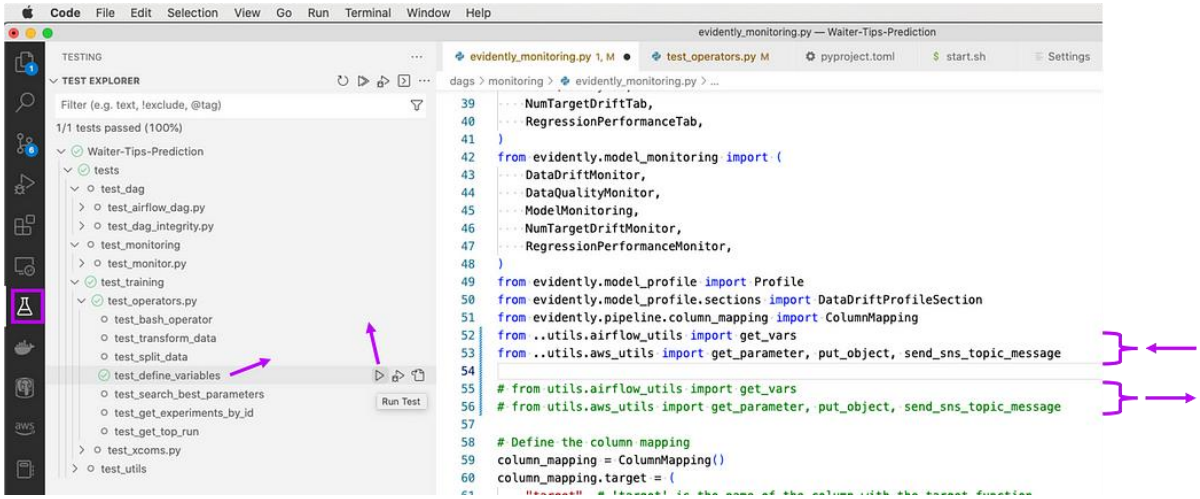



Exhibit-7: Pytest at Work (Image by Author)

I didn't have a chance to look into why the absolute imports didn't work. The solution seems highly relative! Although relative imports worked with *Pytest*, I would not suggest using them in production. Once the code has passed the tests, we can revert to the absolute paths.

It is possible to skip tests on a missing import using pytest.importorskip at the module level. However, this will skip all tests in the module if some import is missing.

If anything we have done so far still doesn't help fix the discovery issue, we still have the command-line option. We can use the pytest command on the terminal to run the tests.

**Part 6: Sorting Imports with isort, Formatting the Code with Black, and Linting with Pylint**

In the previous part, we started to test our code and accomplished the unit testing with *Pytest* and Apache *Airflow* testing. Still do exist a few checks that we can do using *isort*, *Black*, and *Pylint*. They are the topics of this article, which need to be explored as to what they are doing.

Image by storyset on Freepik

Let us start with *isort*.

isort

*isort* is a Python library that groups imports by type, and sorts them alphabetically. In doing so, it follows the PEP 8 standards.

Install *isort*:

pip install isort

We can sort the libraries in two ways. First, we install the Python extension in the Visual Studio Code (VSC) and then right-click on a file open in the editor and choose “Sort Imports” as seen in Exhibit 1:

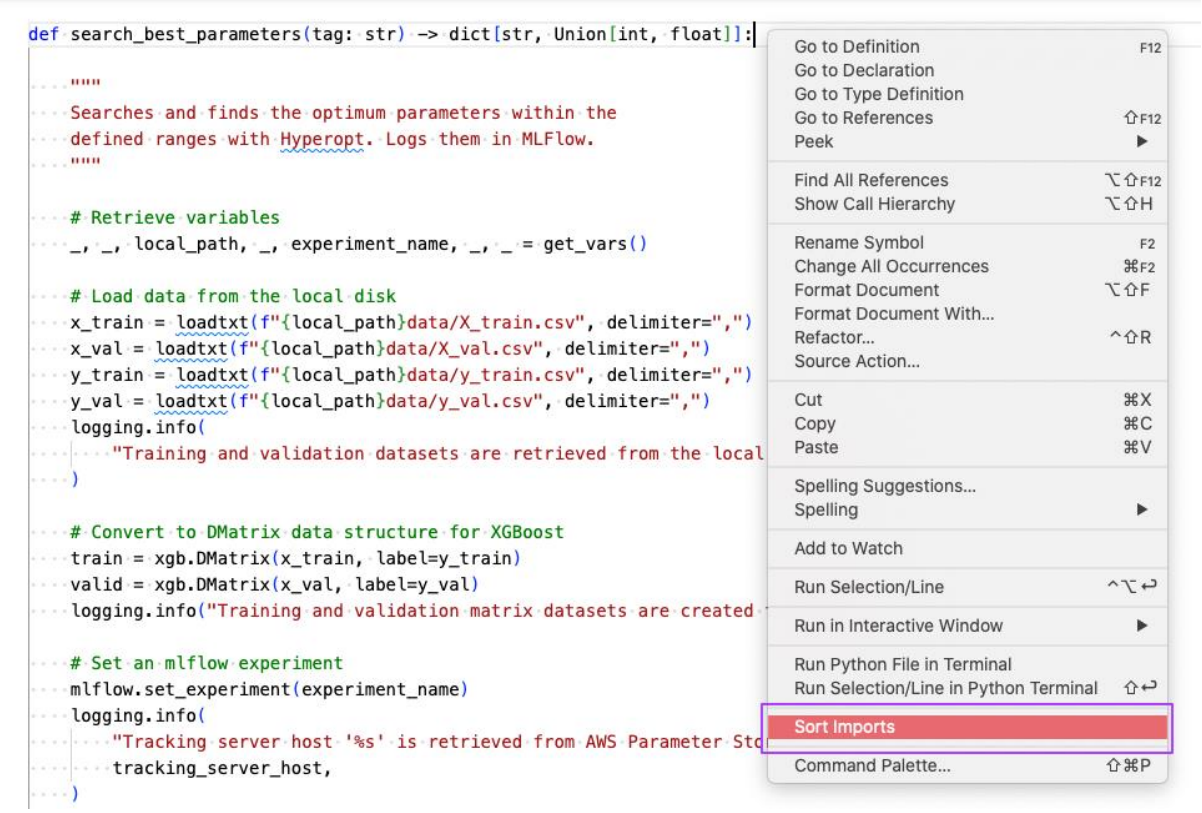



Exhibit-1: Sorting Imports (Image by Author)

Alternatively, we can execute the following command on the terminal of the directory where the module resides, which we want to sort its imports:

isort data_processor.py data_preprocessor.py

We sort the libraries in the data_processor.py and data_preprocessor.py modules. To sort imports in all files in the directory recursively:

isort .

If we want to keep the order convention of *Black* for the imported libraries, we run the following command:

isort . --profile=black

For this, we must have *Black* installed first. Another way of introducing the *Black* profile without mentioning it in the bash command every time we run *isort* is to add it to the pyproject.toml file as follows:

[tool.isort]
profile="black"

According to PEP 518, pyproject.toml is a configuration file to store build system requirements for *Python* projects.

Then, *isort* sorts libraries as specified in the PEP 8 document[1]:

*Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.*
*Imports should be grouped in the following order:*

*1. Standard library imports.*

*2. Related third party imports.*

*3. Local application/library specific imports.*

*You should put a blank line between each group of imports.*

The output will be like this for data_processor.py:


```
# Import libraries
import json
import logging
import pickle
from datetime import datetime
from typing import Any, Literal, Union

import mlflow
import numpy as np
import xgboost as xgb
from hyperopt import STATUS_OK, Trials, fmin, hp, tpe
from hyperopt.pyll import scope
from mlflow.tracking import MlflowClient
from numpy import loadtxt
from sklearn.metrics import mean_squared_error

from utils.airflow_utils import get_vars
from utils.aws_utils import create_parameter, get_parameter, send_sns_topic_message
from utils.mlflow_utils import (
    delete_version,
    get_best_params,
    get_latest_version,
    list_experiments,
    load_models,
    register_model,
    search_runs,
    transition_to_stage,
    update_model_version,
    update_registered_model,
    wait_until_ready,
)
```

Exhibit-2: Sorted Imports (Image by Author)

Here, *isort* didn't insert a blank line above the local application imports though it did correct grouping and sorting and conformed to the order of *Black*.

If we need *isort* to skip files, we should use the skip method. To skip multiple files, we should specify twice: — skip file1 — skip file2. "Values can be file names, directory names, or file paths. To skip all files in a nested path use — skip-glob."[2]

Or by adding the following piece of code in the pyproject.toml file, we can make *isort* skip files without specifying them in the bash command:

[tool.isort]
skip = [".gitignore", ".dockerignore"]

Black

After we organized our imports neatly, we came to format the code. We need to do this according to PEP 8 standards again. Here, *Black* comes into play to help automatically do it. "*Black* is the uncompromising *Python* code formatter."[3]

Install *Black*:

pip install black

We can execute the following command to format the file:

black data_processor.py

If we need to format multiple files in a directory, we can run the following command on the directory's terminal:

black .

This will recursively format all files in that directory. Alternatively, we can write:

black /home/ubuntu/app/Waiter-Tips-Prediction/dags/training/

to achieve the same purpose.

An alternative way to use *Black is to* install the *Black Formatter* extension in VSC. *Black Formatter* is a VSC extension with support for the black formatter. It will eventually replace the black formatting functionality of the *Python* extension. Therefore, we had better use the extension, not the functionality.

After installing the extension, we right-click on a *Python* file, choose "Format Document With...", and then choose "Black Formatter":

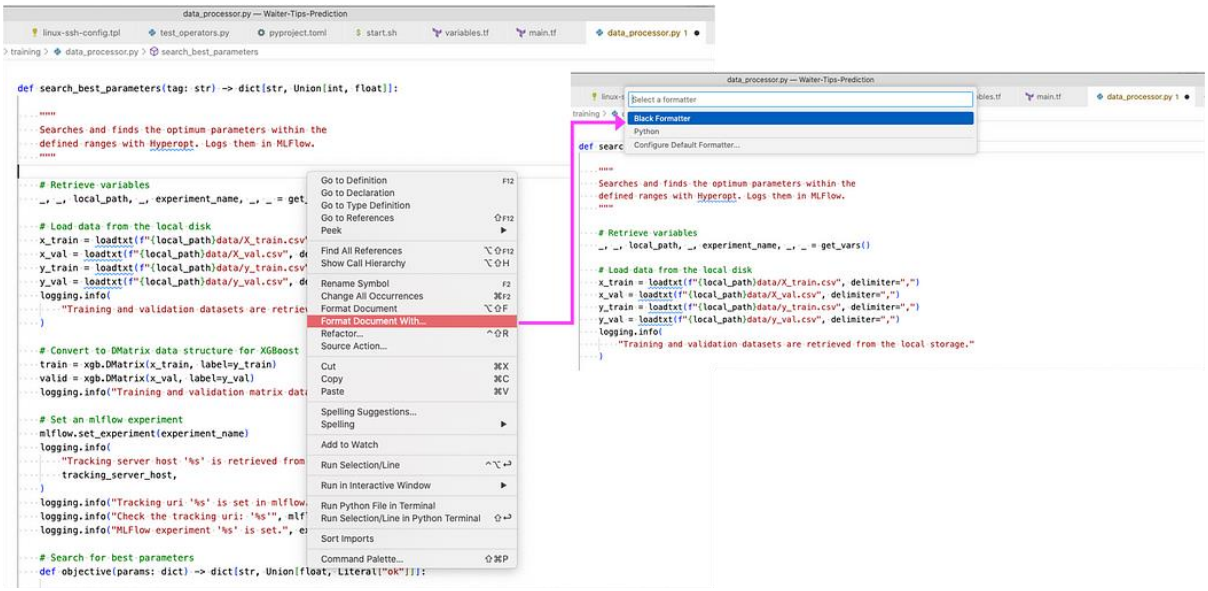


Exhibit-3: VSC Black Formatter Extension (Image by Author)

To make *Black Formatter* the default formatter for *Python* files in VSC, we open the settings and set the "Default Formatter" as *Black*.

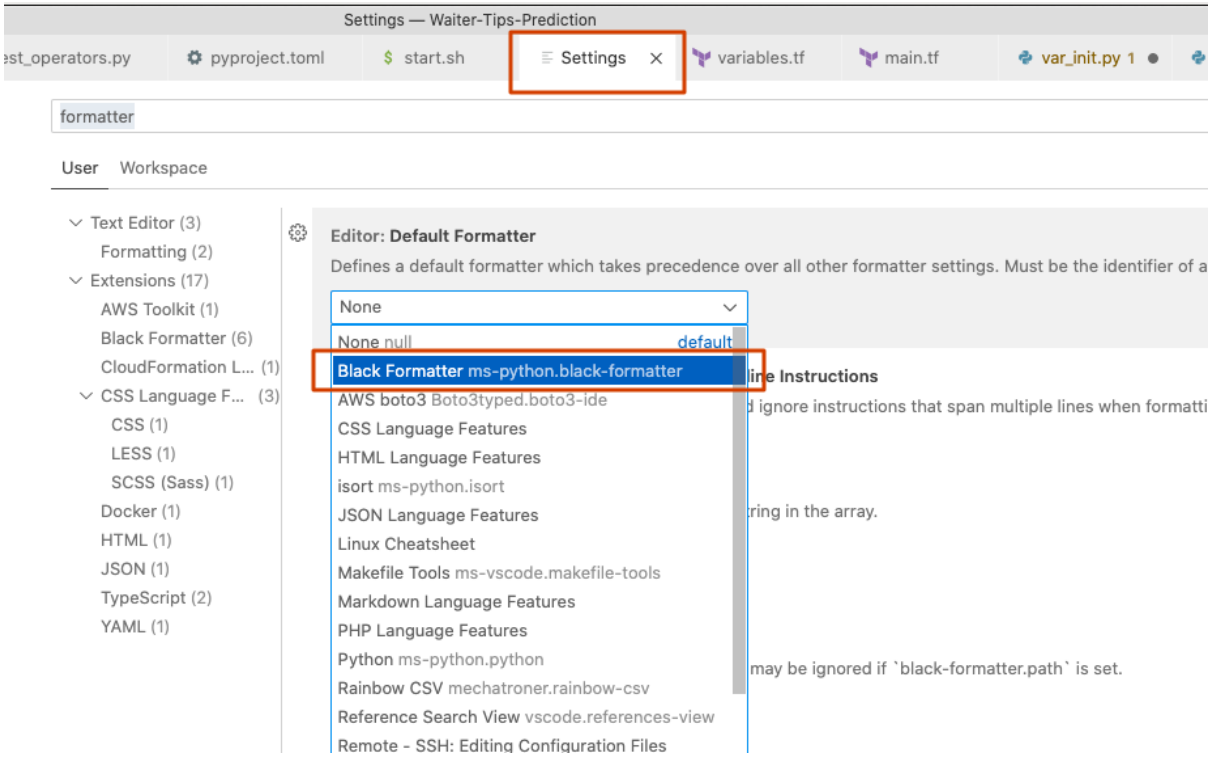



Exhibit-4: Black Formatter as the Default Formatter (Image by Author)

*Black Formatter* will take precedence over all other formatter settings. Then, we mark the "Format on Save" as checked so that *Black* will format the code when saving the file. I think this is a good practice to avoid any technical burden later.

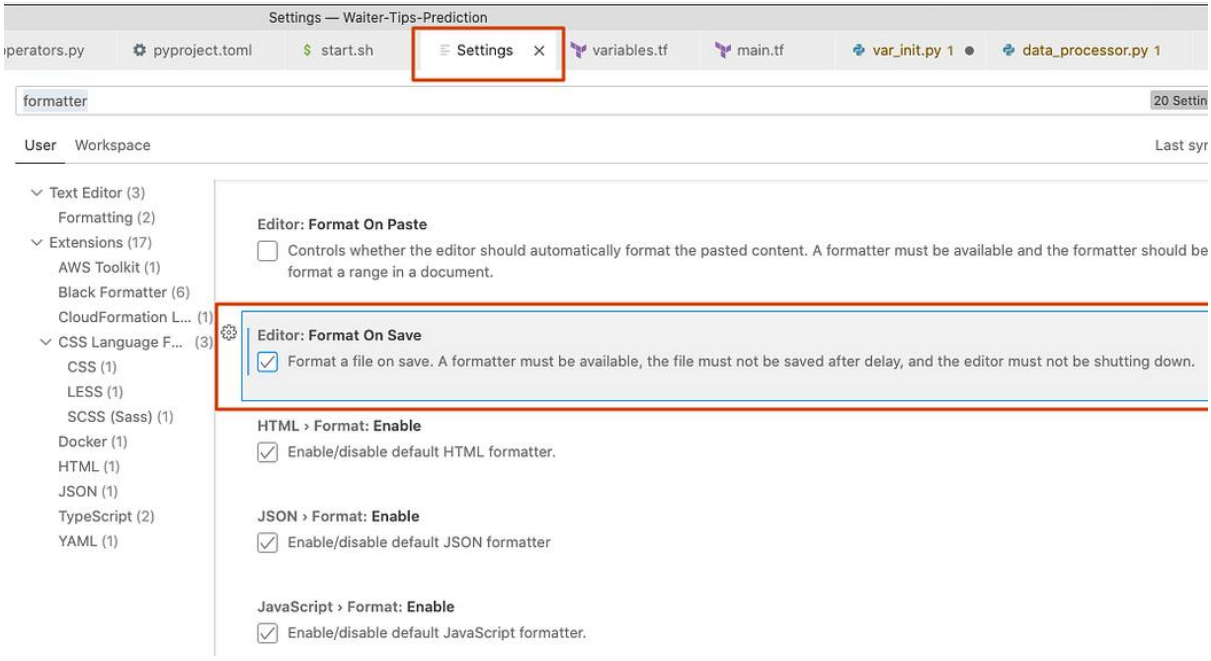



Exhibit-5: Format on Save (Image by Author)

While doing this, we should set *Python*'s black formatting as none.

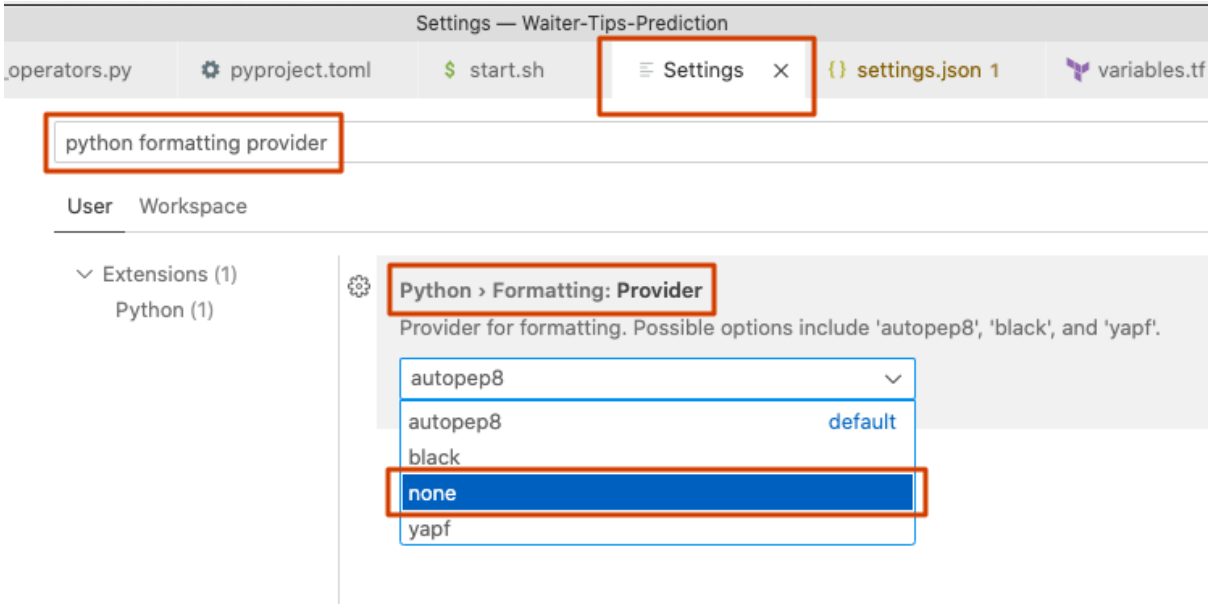



Exhibit-6: Python's Black Formatter (Image by Author)

It's also possible to make the modifications displayed in Exhibits 4 and 5 in User Settings. We first bring the Command Palette by clicking Ctrl+Shift+P (Cmd+Shift+P on Mac) in VSC, type "User Settings," and choose "Preferences: Open User Settings (JSON)."

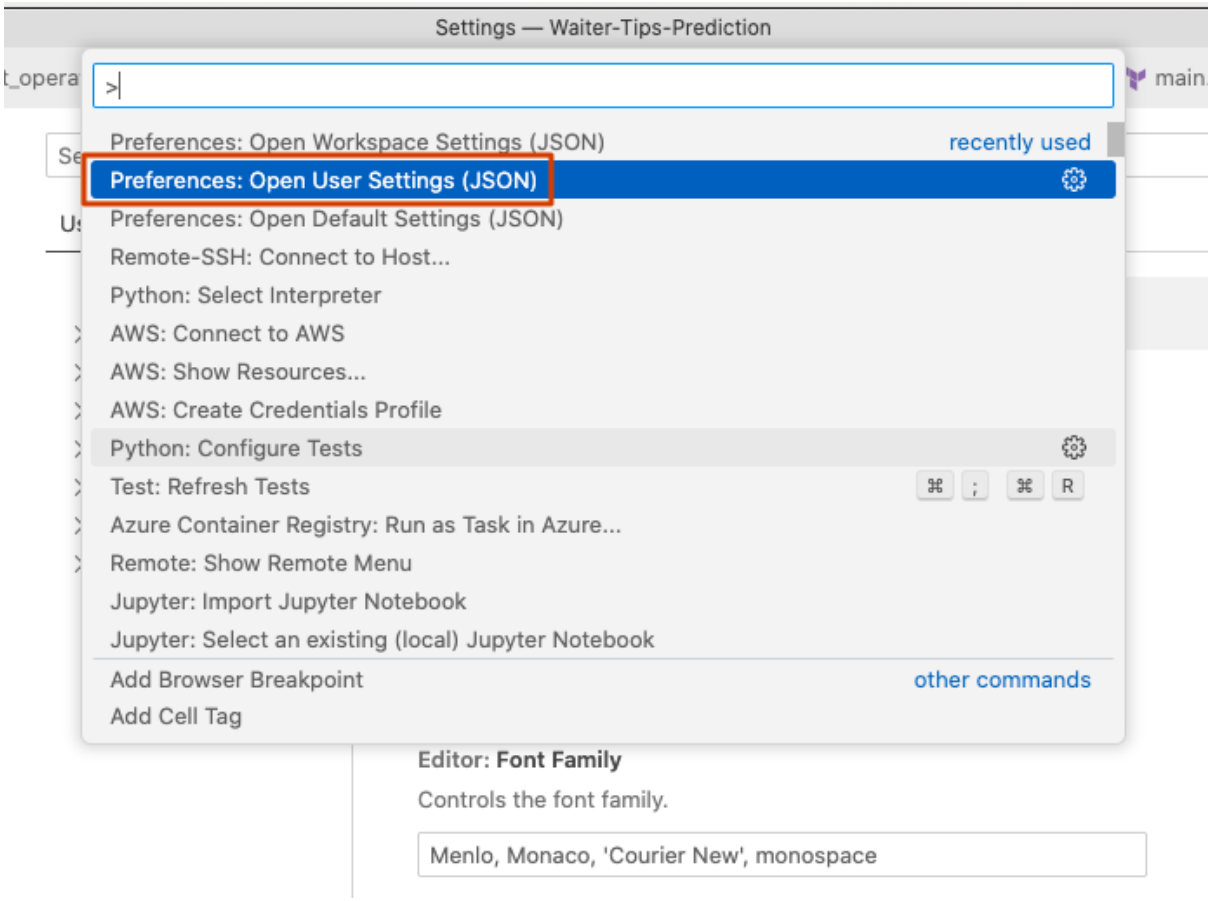



Exhibit-7: User Settings (Image by Author)

Next, the settings.json file will open. We append the following arguments at the end of the file:

"editor.formatOnSave": true,
"editor.defaultFormatter": "ms-python.black-formatter",
"python.formatting.provider": "none"

If we did as shown in Exhibits 4 and 5, these statements would already be in the file.

We can specify our preferred defaults in the settings.json. For example, the line length can be determined as follows:

"black-formatter.args": ["--line-length", "120"]

*Black* will format the code lines according to a maximum of 120 characters. Another way of setting our preferences as default is to specify them in pyproject.toml file. We add the following line in the file:

[tool.black]
line-length = 120
target-version=["py39"]

We also tell *Black* here to run under *Python* version 3.9.

One of the format corrections made by *Black,* as an example, can be seen in Exhibit 8:

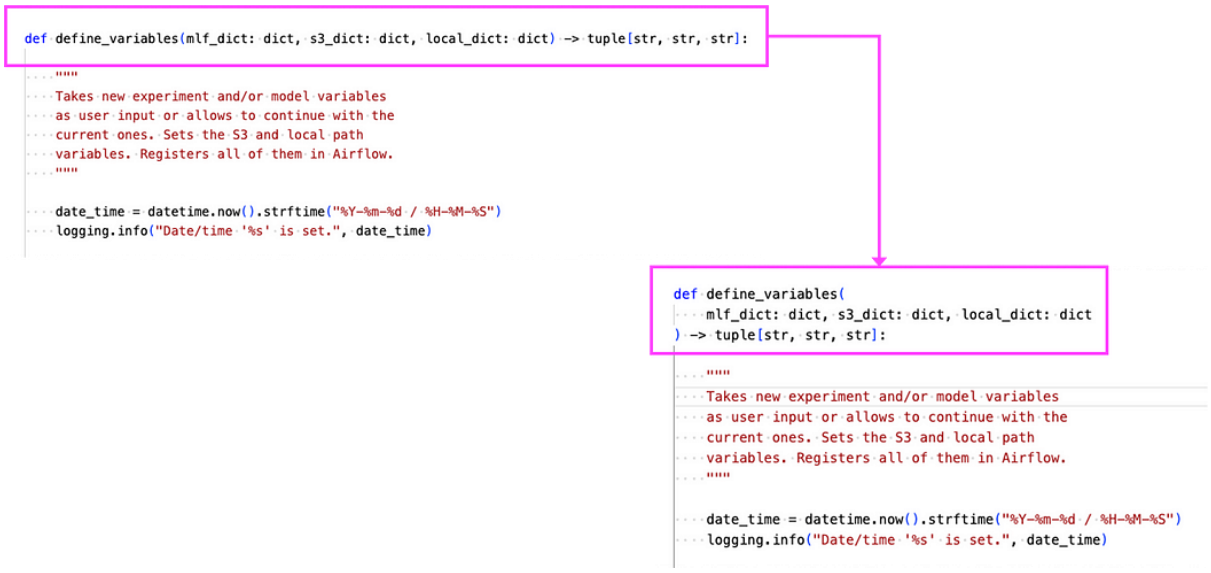



Exhibit-8: Formatting with Black (Image by Author)

Some other corrections that *Black* made were to use double quotes instead of single quotes, choose %-formatting instead of f-Strings, and insert double blank lines between functions or between a callable and the rest of the code. One may refer here to see what other styling *Black* does and the arguments we can pass on the command line or settings file.

*Skipping Files by Black*

We may describe two ways to exclude a file from *Black*'s inspection. The first occurs on the command line.

black . --exclude evidently_monitoring.py

*If* --exclude *is not set,* Black *will automatically ignore files and directories in* .gitignore *file(s), if present.[4]*

The second way of skipping is through pyproject.toml:

```
[tool.black]
force-exclude = "evidently_monitoring.py"
```

We had better use force-exclude in pyproject.toml instead of exclude , as the latter might not skip the file of concern.

After we finish the code styling and formatting, we can proceed to a “deeper” analysis of our code, which we’ll realize with *Pylint*.

Pylint

*Pylint*, unlike dynamic program analyzers, analyzes the code without executing it, thus making it a static program analyzer. It checks for errors, carries out a coding standard, and searches for underlying problems. *Pylint* also helps improve the internal structure of the code without modifying its external behavior. In other words, *Pylint* offers suggestions about improving the code’s design, layout, and implementation while preserving the original functionality[5].

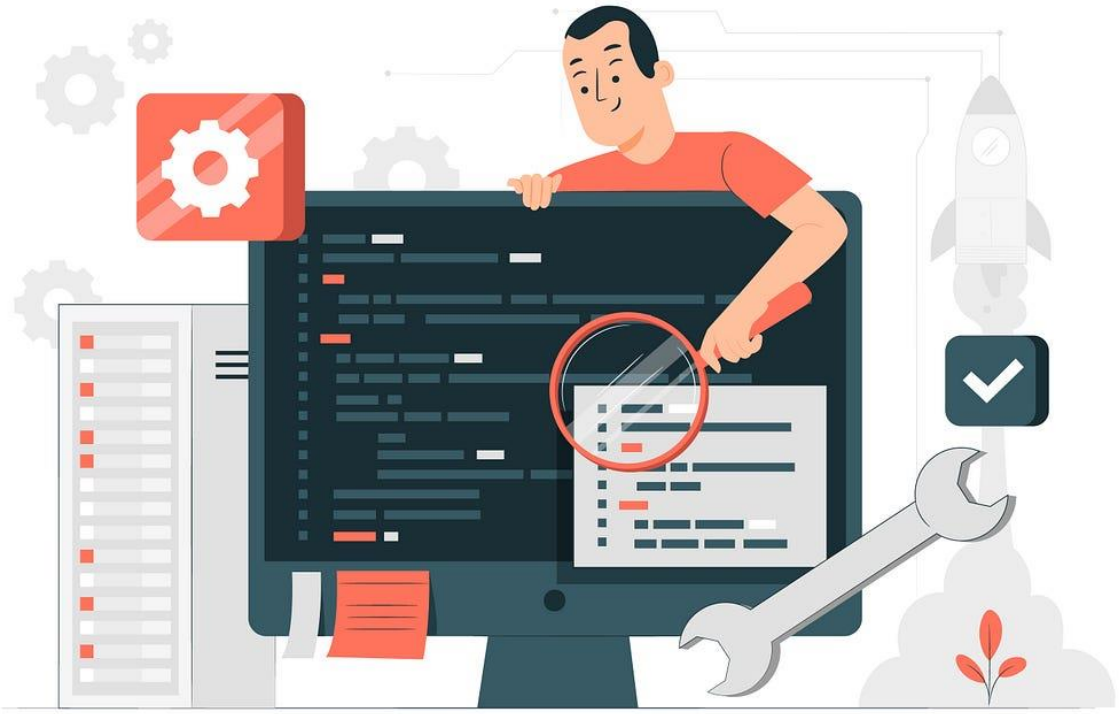


Image by storyset on Freepik

Pylint can be configured to write plugins to add checks for our libraries[6]. However, we don’t think it would be necessary for this work as *Pylint*’s current ecosystem of plugins is quite enough.

Install *Pylint*:

```
pip install pylint
```

To check one or more files, by executing the following command on the terminal of the directory of the file(s) we want to test:

pylint data_processor.py data_preprocessor.py

To check all files in the directory's terminal:

pylint --recursive=y .

Pylint checks a long list of errors. Each error has a name, code, and description. For example, "empty-docstring (C0112)" is used when a module, function, class, or method has an empty docstring. "multiple-imports (C0410)" is used when an import statement importing multiple modules is detected[7]. We may want to avoid raising some of these errors. In that case, we can run *Pylint* as follows:

pylint --disable=C0303,C0413,E0401,C0413 data_processor.py

In the above line, *Pylint* will check all errors in the data_processor.py file except C0303, C0413, E0401, and C0413.

Alternatively, we can use *Pylint* in VSC. To enable it, open the Command Palette by clicking Ctrl+Shift+P (Cmd+Shift+P on Mac) and select the "Python: Select Linter" command.

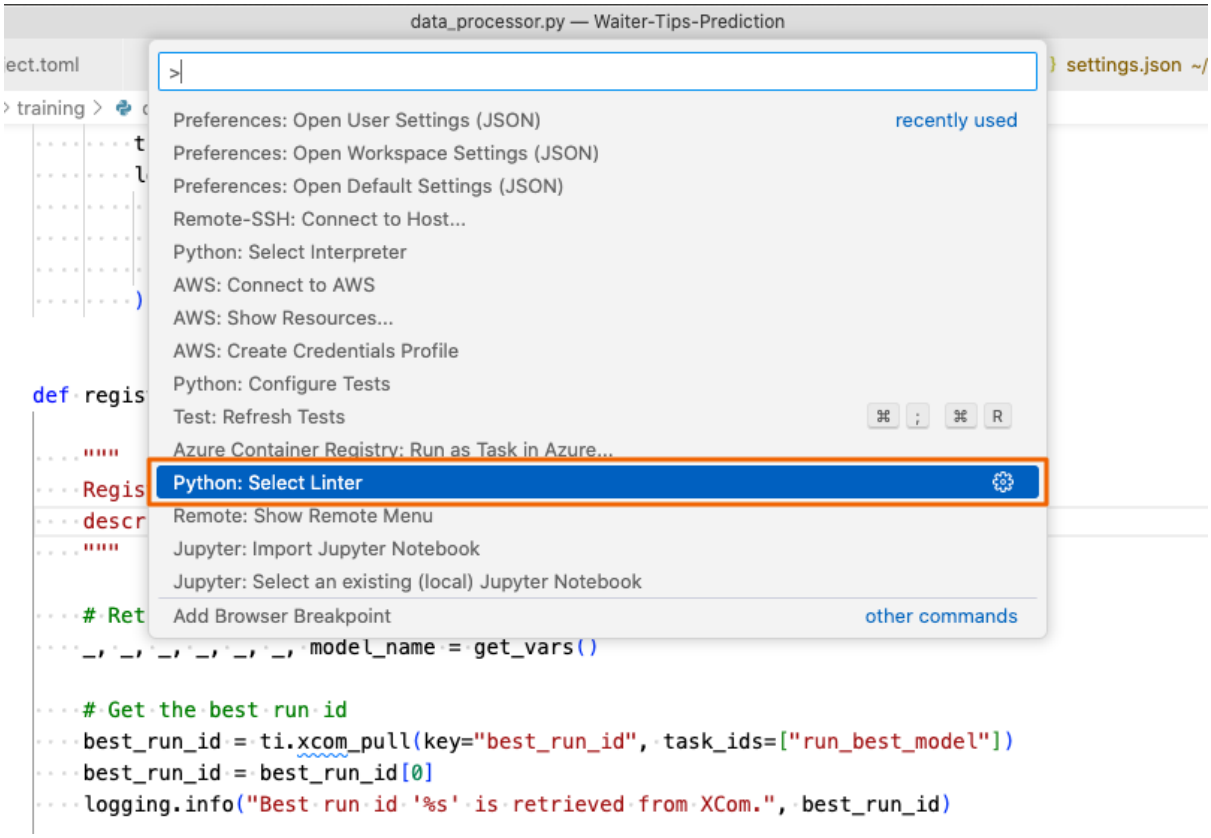



Exhibit-9: Select Linter (Image by Author)

Then, select pylint:

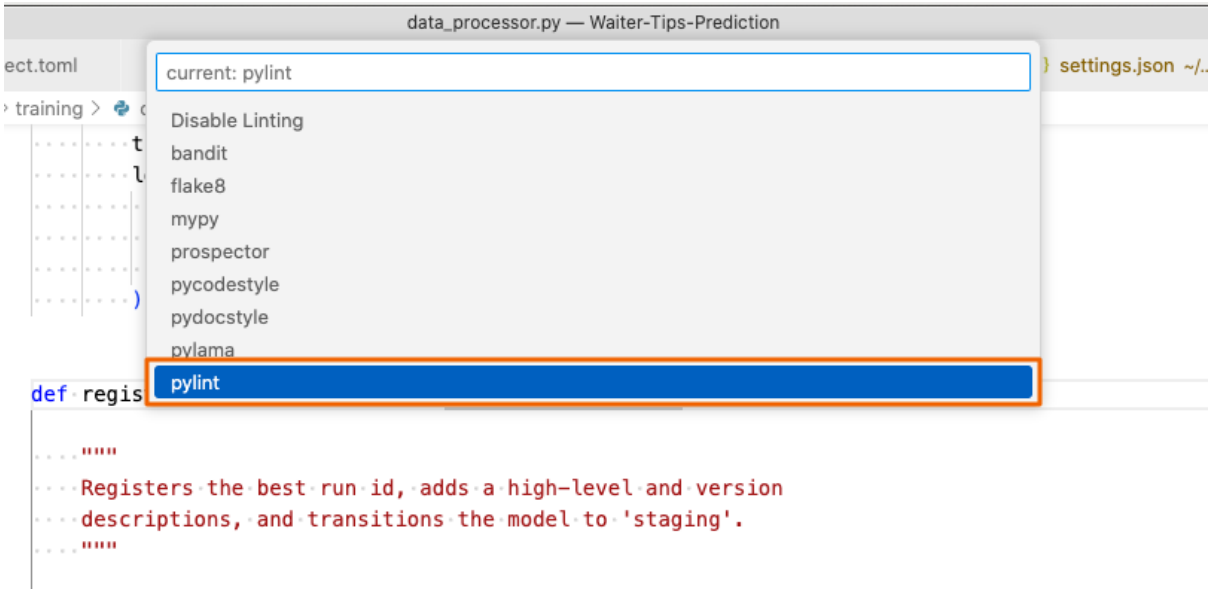



Exhibit-10: Pylint as the Linter (Image by Author)

*The Select Linter command adds* "python.linting.<linter>Enabled": true *to your settings, where* <linter> *is the name of the chosen linter.[8]*

Enabling *Pylint* in VSC will append "python.linting.pylintEnabled": true in settings.json. We have seen how to open this file before.

"python.linting.pylintEnabled": true

*Enabling a linter prompts you to install the required packages in your selected environment for the chosen linter.[8]*

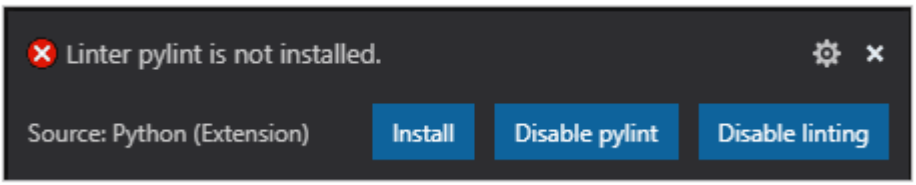



Exhibit-11: Install Prompt for Pylint (Taken from https://code.visualstudio.com/docs/python/linting)

To run linting, open the Command Palette by clicking Ctrl+Shift+P (Cmd+Shift+P on Mac) and select the "Python: Run Linting" command.

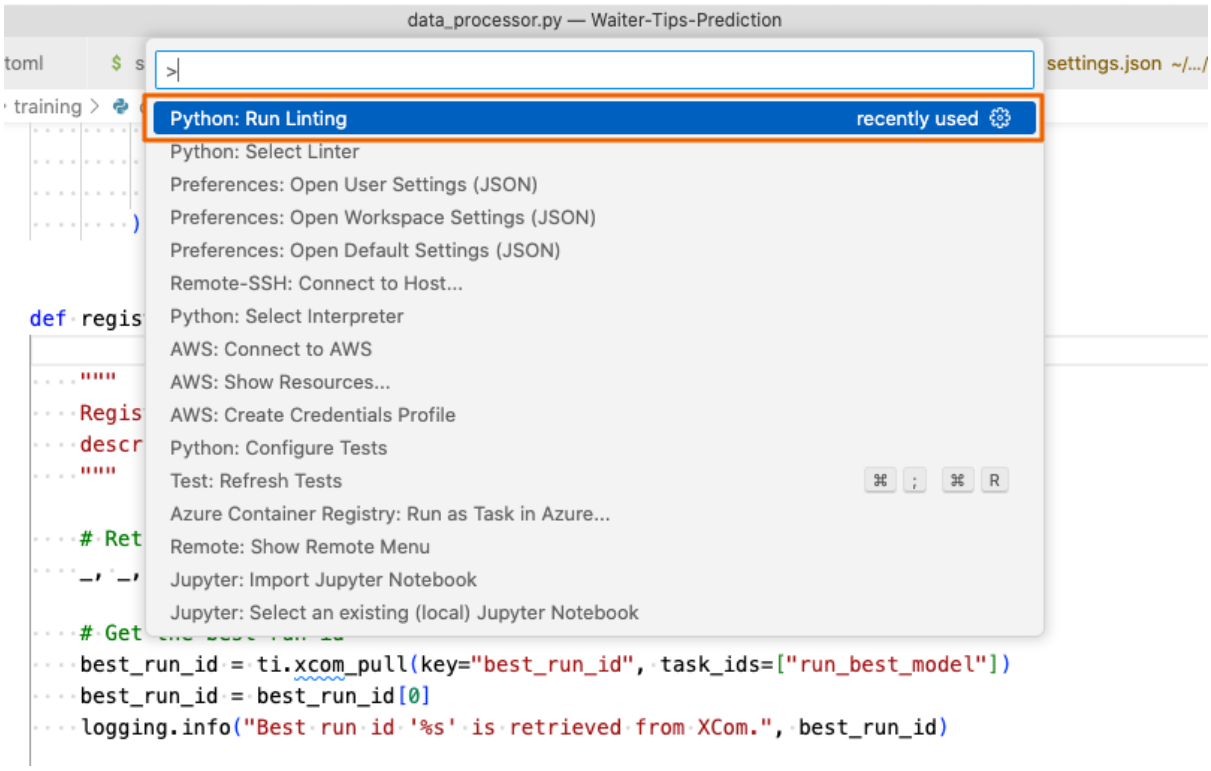



Exhibit-12: Run Pylint (Image by Author)

*Linting will run automatically when you save a file. Issues are shown in the Problems panel and as wavy underlines in the code editor. Hovering over an underlined issue displays the details:[8]*

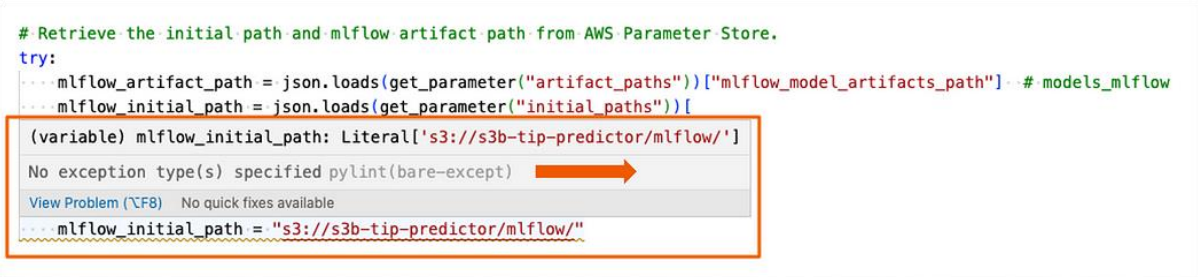



Exhibit-13: Error Details by Python's Pylint (Image by Author)

When we enabled *Pylint* in VSC, we also had the "Lint On Save" option checked. Thus, whenever we save a *Python* file, it will be linted.

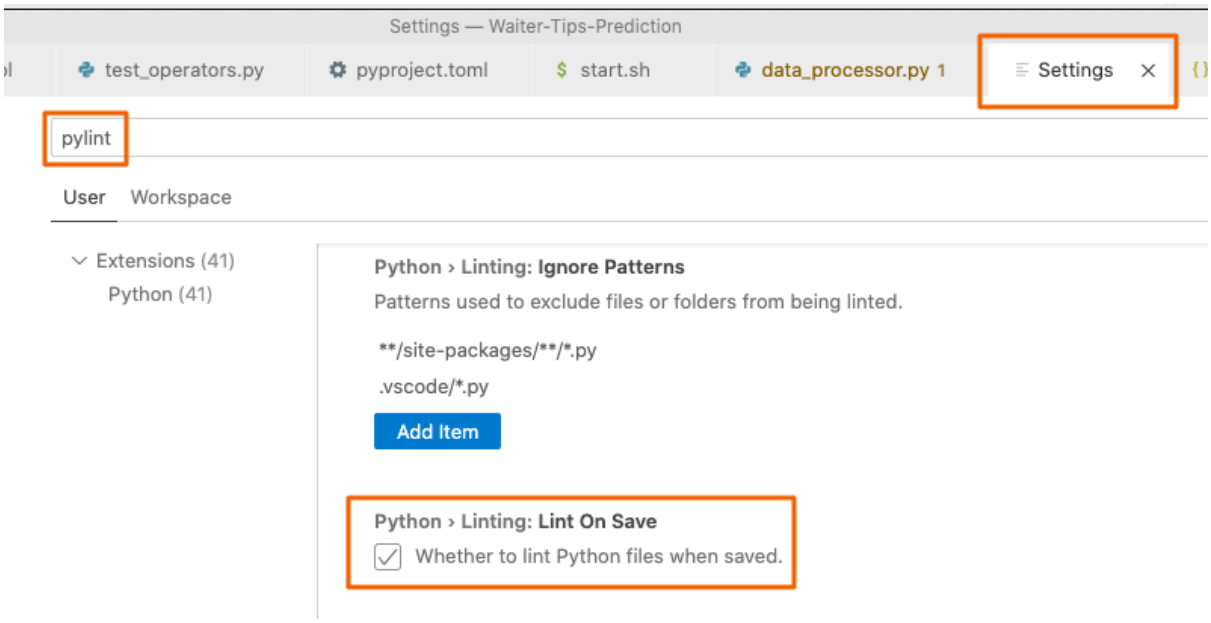



Exhibit-14: Lint On Save (Image by Author)

That has been *Python*'s pylint functionality so far. Another way to put *Pylint* into effect is to install the *Pylint* extension in VSC. For this, we first need to disable *Python*'s pylint functionality in VSC's Settings:

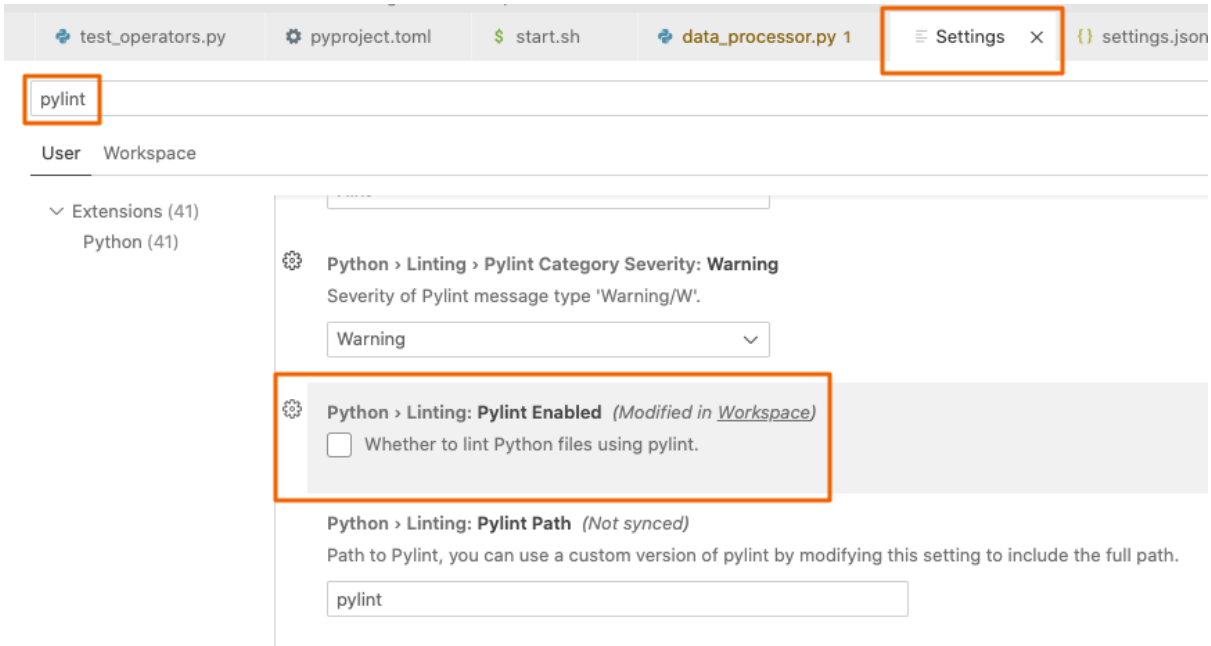



Exhibit-15: Disabling Python's Pylint Functionality (Image by Author)

This operation will remove "python.linting.pylintEnabled": true from settings.json. Then, we install the *Pylint* extension. The result is as below:

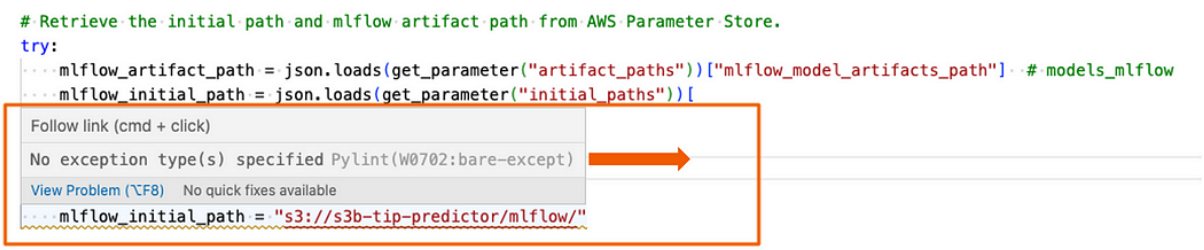



Exhibit-16: Error Details by Pylint Extension (Image by Author)

Different from *Python*'s pylint functionality, the *Pylint* extension also returns the error code.

When we execute pylint on the terminal, we get the output:

pylint data_processor.py

********** Module dags.training.data_processor
data_processor.py:79:0: W0702: No exception type(s) specified (bare-except)

---
Your code has been rated at 9.95/10 (previous run: 9.95/10, +0.00)

As you see, to get this output, we don't need to execute the code. As mentioned earlier, *Pylint* is a static code analyzer. It analyzes the code without actually running it.

We can specify our preferred defaults in settings.json for the *Pylint* extension:

"pylint.args": ["--rcfile=/home/ubuntu/app/Waiter-Tips-Prediction/.pylintrc"]

.pylintrc is a configuration file that contains arguments we pass, and instructs *Pylint* to run by them. No matter how we install and use *Pylint*, we can use .pylintrc. One may refer here to get more information about what it is. For example, we can create a highly simple .pylintrc to ask *Pylint* to skip two types of errors during checking:

[MESSAGES CONTROL]

disable=
  import-error,
  wrong-import-position,

An alternative to .pylintrc is pyproject.toml file as usual. Therefore, we can pass our arguments as follows:

[tool.pylint.messages_control]
disable = [
  "import-error",
  "wrong-import-position",
  "import-outside-toplevel",
  "too-many-locals",
  "invalid-name",
  "pointless-statement",

```
  "redefined-outer-name",
  "broad-except",
  "line-too-long"
  ]

[tool.black]
line-length=80
target-version=["py39"]

[tool.isort]
profile="black"
```

I prefer pyproject.toml as the configuration file, as we can describe all settings for various tools and packages in one place.

The disable variable should raise some concerns and not be too crowded, or there will be no point in running tests. We do not know about hidden errors in the code initially. Running the tests after writing up the entire code may unveil many errors. If we are reluctant to fix some of them, we would be constrained to fill the disable variable in the pyproject.toml file. That shouldn't be a good practice. Therefore, we should know that the safe and sound course of action is to run the tests on a manageable size of code at each step towards the final total unless we want to get into depression seeing so many errors later. We should take care of all errors before committing the project to Git.

Even though having said so, we may still need to accept some errors! For example, *Pylint* returns an "import-outside-toplevel" error when an import statement is used anywhere other than the module top level[7]. We may not want to eliminate this error as *Airflow* doesn't like it. *Airflow* suggests local imports. We should consider priorities. We don't let *Airflow* take the heavy load, so we may ask *Pylint* to turn a blind eye to this error by adding it in the disable variable.

Both .pylintrc and pyproject.toml (whichever you decide to use, though) should be in the root directory of your project, which is /home/ubuntu/app/Waiter-Tips-Prediction.

*Skipping Files by Pylint*

The first option to skip a file is by the command line argument:

```
pylint --ignore=data_processor.py
```

*Pylint* skips data_processor.py. Files or directories to be skipped should be base names, not paths. That's why it should be — ignore=data_processor.py instead of — ignore=/home/ubuntu/app/Waiter-Tips-Prediction/dags/training/data_processor.py.

The second option is through the pyproject.toml file:

```
[tool.pylint.MASTER]
# Add files or directories matching the regex patterns to the blacklist. The
# regex matches against base names, not paths.
ignore-patterns = "data_processor.py"
```

Don't be confused by the "base names" clause in the comments. It accepts paths with wildcards when necessary to ignore subdirectories or files with a particular extension.

We should do as sufficient docstring, typing, and annotation as possible. It is perhaps the most boring job today, but certainly, the most beneficial measure tomorrow.

If you want to execute the testing tools separately, I suggest you follow the order: *isort*, *Black*, *Pylint*, *Pytest*. However, the more practical way is to implement them together from a central application. *pre-commit* accomplishes this objective. Another area yet unchecked is the cloud infrastructure. *LocalStack* inspects if the cloud resources function correctly. The following article will discuss these two tools and complete the testing phase.

**Part 7: Identifying Issues Before Submission with pre-commit, and Simulating AWS Services with LocalStack**

ered to inspect the code and identify issues when committing a project to *GitHub*. It raises the issues identified by libraries such as *Black*, *isort*, and *Pytest*, which are configured in a document called .pre-commit-config.yaml located in the project folder (/home/ubuntu/app/Waiter-Tips-Prediction).

In the YAML configuration file, we specify a list of hooks so that pre-commit installs and executes them (written in any language) before every commit[1].

**Installation**

Install pre-commit:

pip install pre-commit

Check the installation:

pre-commit --version

**Configuration**

Create a .pylintrc file in the project folder. We have seen it in the previous article. In this file, we specify the errors we ask pylint to ignore during pre-commit. Populate it as follows:

```
[MESSAGES CONTROL]
disable=
  import-error, # E0401
  wrong-import-position, # C0413
  import-outside-toplevel, # C0415
  too-many-locals, # R0914
  invalid-name, # C0103
  pointless-statement, # W0104
  redefined-outer-name, # W0621
  broad-except, # W0703
  line-too-long, # C0301
  too-many-statements, # R0915
  no-name-in-module, # E0611
  duplicate-code # R0801
```

Each error has a code name as specified in comments[2].

Create a .pre-commit-config.yaml file in the project folder.

```
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v3.2.0
  hooks:
    - id: trailing-whitespace
    - id: end-of-file-fixer
    - id: check-yaml
- repo: https://github.com/pycqa/isort
  rev: 5.10.1
  hooks:
    - id: isort
      name: isort
      entry: isort
      language: python
      types: [python]
- repo: https://github.com/psf/black
  rev: 22.6.0
  hooks:
    - id: black
      name: black
      entry: black
      language: python
      language_version: python3.9
      types: [python]
- repo: local
  hooks:
    - id: pylint
      name: pylint
      entry: pylint
      language: python
      types: [python]
      args: [
        "-rn", # Only display messages
        "-sn", # Don't display the score
        "--rcfile=.pylintrc",
        "--ignore=E0401, C0413, C0415, R0914, C0103, W0104, W0621, W0703, C0301, R0915,
E0611, R0801",
        "--recursive=y"
      ]
- repo: local
  hooks:
    - id: pytest-check
      name: pytest-check
      entry: pytest
      language: system
      pass_filenames: false
      always_run: true
      files: ^tests/
```

Let us take a closer look at the components of the .pre-commit-config.yaml file.

repos is a list of repository mappings. Each hook has the resource repo with the revision or tag. The repository mapping tells *pre-commit* where to get the code for the hook from.

```
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v3.2.0
  hooks:
```

repo is the repository URL to git clone from. *pre-commit* will get the code from this repo. Sometimes, the repo argument tells *pre-commit* to run the executable, not from the git repository but the local system. We'll see this a little later.

rev is the revision or tag to clone. For example, *pre-commit* uses version 22.6.0 of black. However, we can substitute this version for another one contained in the black repo.

hooks is a list of hook mappings. It describes the test to be done.

Each hook is identified with an id and name. id refers to which hook from the repository to use. name overrides the name of the hook, and is shown during hook execution.

```
hooks:
   - id: trailing-whitespace
   - id: end-of-file-fixer
   - id: check-yaml
```

In the above snippet are some out-of-the-box hooks for *pre-commit*. For example, trailing-whitespace trims trailing white spaces. One can refer to the documentation to find out more about pre-commit-hooks.

Similarly, *pre-commit* supports hooks provided by other libraries, such as:

```
hooks:
   - id: pylint
     name: pylint
```

According to these lines, .pre-commit-config.yaml will expose our code to *Pylint* inspection.

```
entry: black
language: python
language_version: python3.9
types: [python]
```

entry is the entry point for the executable to run. Here, black will be run to format the code.

language is the language of the hook, which shows *pre-commit* how to install the hook.

language_version overrides the language version for the hook if we want to run the hooks on a specific version of the language. “For each language, they default to using the system-installed language (So for example, if I’m running python3.7 and a hook specifies *python*, *pre-commit* will run the hook using python3.7). Sometimes you don’t want the default system-installed version, so you can override this on a per-hook basis by setting the language_version.[1]” The above segment tells *pre-commit* to use python3.9 to run the black hook.

types indicates the list of file types to run on and overrides the default file types. In the example given, *pre-commit* will run black on *Python* files.

```
- repo: local
  hooks:
    - id: pylint
      name: pylint
      entry: pylint
      language: python
      types: [python]
      args: [
        "-rn", # Only display messages
        "-sn", # Don't display the score
        "--rcfile=.pylintrc",
        #"--ignore=E0401, C0413, C0415, R0914, C0103, W0104, W0621, W0703, C0301, R0915, E0611, R0801",
        "--recursive=y"
      ]
```

*pre-commit* allows using local sentinel for a repo section, which means it will run pylint as installed on the system instead of from the git repository.

With args , we can pass static arguments to the hook in our YAML configuration file.

— rcfile, by referring to .pylintrc we created earlier, asks *pre-commit* to take the instructions in that file into consideration when running pylint. In .pylintrc , we ask pylint to ignore particular errors. An alternative way to using — rcfile to ignore errors is to use — ignore argument and list the error codes we won't care about.

— recursive=y tells *pre-commit* to run pylint recursively on the project folder.

```
- repo: local
  hooks:
    - id: pytest-check
      name: pytest-check
      entry: pytest
      language: system
      pass_filenames: false
      always_run: true
      files: ^tests/
```

The default value for pass_filenames is True, meaning file names can be passed to the hook. Here, it is False ; so no file names will be passed.

If always_run is True, this hook will run even if no matching files exist.

files indicates the pattern of files to run on. In the configuration, we tell *pre-commit* to run pytest on all files within the "tests" folder but not on any other folder.

Next, initialize git in the project folder:

```
git init
```

*Git* is initialized in the folder in which we run *pre-commit*.

Create a pre-commit hook file (run the command only once):

pre-commit install

This will return:

pre-commit installed at .git/hooks/pre-commit

Check to see if it is created:

ls .git/hooks/

The *pre-commit* file is not committed to *GitHub* and is generated on the local machine. After cloning the repo, any team member should execute the above command to create the *pre-commit* file on his local computer.

**Execution**

Now, we can run each hook manually. -a refers to all files. The run occurs without and before git add and git commit. Don't forget to include the .gitignore file. Hooks don't run on ignored files.

pre-commit run -a

This command will return something like[1]:

[INFO] Initializing environment for https://github.com/pre-commit/pre-commit-hooks.
[INFO] Initializing environment for https://github.com/psf/black.
[INFO] Installing environment for https://github.com/pre-commit/pre-commit-hooks.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
[INFO] Installing environment for https://github.com/psf/black.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
Check Yaml.....Passed
Fix End of Files.....Passed
Trim Trailing Whitespace.....Failed
- hook id: trailing-whitespace
- exit code: 1

Files were modified by this hook. Additional output:

Fixing sample.py

black.....Passed

As the report states, a non-zero exit code denotes "failure." The error is trailing-whitespace , and is corrected by removing the trailing white spaces from the file.

We can run the git diff command to see changes made in the file(s).

To run a particular hook (e.g., pytest-check) on all files:

pre-commit run pytest-check -a

What we have done so far was to manually execute *pre-commit* on the command line. An alternative, even better way, is to use the trigger mechanism.

When we stage and commit the files to *GitHub*,

```
git add .
git commit -m 'initial commit'
```

the above command triggers the *pre-commit* to run hooks first. It inspects the snapshot that's about to be committed. It helps us check for code style (e.g., *Pylint*), trailing whitespace (the default hook does exactly this), or appropriate documentation on new methods. Therefore, *pre-commit* enables us to notice if we have forgotten something, to make sure tests run, or to examine whatever we need to check and identify in the code. Exiting non-zero from this hook aborts the commit[3]. In other words, if the code doesn't pass all tests, we won't be able to commit it in *GitHub*.

When committing the files in the usual manner, if the *pre-commit* exits non-zero and aborts the commit, we need to get back to the code and make the corrections as shown on our screen.

If you want to bypass tests while committing the files, run:

```
git commit --no-verify -m "skip commit hooks in git and commit this"
```

To summarize, *pre-commit* lifts the burden of applying tests to our code one at a time off our shoulders. It facilitates our job by implementing all inspection libraries and allows committing to *GitHub* if the code passes checks successfully.

**Last Word on Testing Tools**

We perform the tests and inspections during the code development, and they are not for use in production. As usual, while working in the virtual environment to develop the code, we should install the inspection libraries in the dev mode, such as:

```
pipenv install --dev pre-commit
```

In this project, all files will be installed in Pipfile (if pipenv). The final version of our Pipfile looks like this:

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
numpy = "*"
pandas = "*"
plotly = "*"
plotly-express = "*"
mlflow = "*"
jupyter = "*"
scikit-learn = "*"
seaborn = "*"
hyperopt = "*"
```

```
xgboost = "*"
fastparquet = "*"
boto3 = "*"
statsmodels = "*"
pickle5 = "*"
jupyter-contrib-nbextensions = "*"
astroid = "*"
psycopg2-binary = "*"
pendulum = "*"
werkzeug = "==2.1.2"
flask = "*"
evidently = "*"

[dev-packages]
pylint = "*"
black = "*"
isort = "*"
pytest = "*"
pre-commit = "*"
localstack = "*"

[requires]
python_version = "3.9"
```

As seen, we installed all testing-related packages in the dev mode.

Code testing is done. Now remains the final testing for the cloud infrastructure.

**LocalStack**

*LocalStack* is a cloud service emulator that runs in a single container on a laptop or in a CI environment. It enables running AWS applications or Lambdas entirely on a local machine without connecting to a remote cloud provider[4].

Image by macrovector on Freepik

*LocalStack* runs in a Docker container, so *Docker* should be installed before. We can start *LocalStack* on our local machine, as a *Docker* container on our machine, or even on a remote *Docker* host. We prefer the first option in this project.

**Installation**

Install *LocalStack*:

python3 -m pip install localstack

Check if it is installed correctly:

localstack --version

It installs the localstack-cli which is used to run the *Docker* image that hosts the *LocalStack* runtime.

We open the *Docker* application.

By default, *LocalStack* is started inside a *Docker* container in the detached mode by running:

localstack start -d

However, I don't suggest the above deployment as we could encounter problems. Instead, we should start *LocalStack* from the *Docker* container. We will work on an Ubuntu machine, but the docker deployment of *LocalStack* is good with Mac and Windows.

You can refer to the Docker page to install its desktop application on Mac. For Ubuntu, we execute the installation commands as described on their page. We will prepare *LocalStack* in the container in three main steps.

*Step I: Deploying Docker*

1. Uninstall older versions:

sudo apt-get remove docker docker-engine docker.io containerd runc

2. Update the apt package index:

sudo apt-get update

3. Install packages to allow apt to use a repository over HTTPS:

sudo apt-get install ca-certificates curl gnupg lsb-release

4. Add Docker's official GPG key:

```
sudo mkdir -m 0755 -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
```

5. Use the following command to set up the repository:

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

6. Install *Docker* Engine, containerd, and *Docker* Compose:

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

7. Verify that the Docker Engine installation is successful by running the hello-world image:

sudo docker run hello-world

*This command downloads a test image and runs it in a container. When the container runs, it prints a confirmation message and exits. We have now successfully installed and started* Docker *Engine. The docker user group exists but contains no users, which is why we're required to use sudo to run* Docker *commands.[5]*

*Step II: Preparing the Container*

1. Next, run the *LocalStack* container:

sudo docker run -d --rm -p 4566:4566 -p 4510-4559:4510-4559 localstack/localstack

2. List the containers created:

sudo docker container ls -a

This returns the following output:

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f7c29a46267e localstack/localstack "docker-entrypoint.sh" 12 seconds ago Up 7 seconds (health: starting) 0.0.0.0:4510-4559->4510-4559/tcp, :::4510-4559->4510-4559/tcp, 0.0.0.0:4566->4566/tcp, :::4566->4566/tcp, 5678/tcp jovial_pare

3. Access the container terminal:

sudo docker container exec -it f7c29a46267e /bin/bash

4. Update the container:

apt-get update

5. Install the editors and wget tool:

apt-get install vim nano wget -y

*Step III: Installing Terraform*

As the final step, we will install *Terraform, tflocal*, and *LocalStack AWS CLI*. *tflocal* is a small wrapper script to run *Terraform* against *LocalStack*. *LocalStack AWS CLI* provides the awslocal command, which is a thin wrapper around the AWS command line interface for use with *LocalStack*.

We'll perform the following steps on the container terminal.

We install *Terraform* as described on their page:

1. "Ensure that your system is up to date and you have installed the gnupg, software-properties-common, and curl packages installed. You will use these packages to verify HashiCorp's GPG signature and install HashiCorp's Debian package repository.[6]"

apt-get update && apt-get install -y gnupg software-properties-common

2. Install the HashiCorp GPG key:

```
wget -O- https://apt.releases.hashicorp.com/gpg | \
  gpg --dearmor | \
  tee /usr/share/keyrings/hashicorp-archive-keyring.gpg
```

3. Verify the key's fingerprint:

```
gpg --no-default-keyring \
  --keyring /usr/share/keyrings/hashicorp-archive-keyring.gpg \
  --fingerprint
```

The gpg command will report the key fingerprint:

/usr/share/keyrings/hashicorp-archive-keyring.gpg
---

pub   rsa4096 2023-01-10 [SC] [expires: 2028-01-09]
      798A EC65 4E5C 1542 8C8E  42EE AA16 FCBC A621 E701
uid         [ unknown] HashiCorp Security (HashiCorp Package Signing)
<security+packaging@hashicorp.com>
sub   rsa4096 2023-01-10 [S] [expires: 2028-01-09]

4. Add the official HashiCorp repository to your system. The lsb_release -cs command finds the distribution release codename for your current system, such as buster, groovy, or sid.

```
echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
  https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
  tee /etc/apt/sources.list.d/hashicorp.list
```

5. Download the package information from HashiCorp:

```
apt update
```

6. Install Terraform from the new repository:

```
apt-get install terraform
```

7. Verify that the installation works:

```
terraform -help
```

That will list Terraform's available subcommands:

```
Usage: terraform [global options] <subcommand> [args]


The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.
...
```

8. Create a folder named "terraform":

```
mkdir terraform
```

9. Install *tflocal*:

```
pip install terraform-local
```

10. Install *awslocal*:

```
pip install awscli-local
```

With awslocal, we can, for example, execute the following command

```
awslocal ec2 describe-vpcs
```

instead of

```
aws --endpoint-url=http://localhost:4566 ec2 describe-vpcs
```

We are finally done with the required installations to run *LocalStack*!

**Stack Creation**

From now on, it won't be much different from *Terraform* itself execution-wise. The difference is that *LocalStack* builds infrastructure virtually while *Terraform* deploys it physically with *AWS*. *LocalStack* turns *AWS* deployment into a "SimCity" playground.

We go into the "terraform" folder we generated earlier, and create a file named main.tf.

```
cd terraform
nano main.tf
```

We can create all the necessary files in this folder to execute for *LocalStack*, as we will run them in *Terraform* to build the real infrastructure. To show the implementation of *LocalStack* with a small example, we'll deploy a VPC and two subnets. Anyone can create any *AWS* resources in *LocalStack* in a similar way. One caveat is that not all *AWS* services are available in *LocalStack*. You may refer to the documentation to see available services and their coverage levels. Still, many services are at our disposal to try at no cost.

In main.tf file, we place the following code to deploy our VPC and subnets:

```
provider "aws" {
 access_key                  = "test"
 secret_key                  = "test"
 region                      = "eu-west-1"
 s3_use_path_style           = true
 skip_credentials_validation = true
 skip_metadata_api_check     = true
 skip_requesting_account_id  = true
}

resource "aws_vpc" "wtp_vpc" {
 cidr_block           = "10.0.0.0/16"
 enable_dns_hostnames = true
 enable_dns_support   = true

 tags = {
  Name      = "wtp-vpc"
  Environment = "production"
 }
}

resource "aws_subnet" "wtp_public_subnet" {
 vpc_id                  = aws_vpc.wtp_vpc.id
 cidr_block              = "10.0.1.0/24"
 map_public_ip_on_launch = true
 availability_zone       = "eu-west-1a"

 tags = {
  Name      = "wtp-pub_subnet"
  Environment = "production"
 }
}
```

```
resource "aws_subnet" "wtp_private_subnet" {
 vpc_id          = aws_vpc.wtp_vpc.id
 cidr_block      = "10.0.2.0/24"
 availability_zone = "eu-west-1b"

 tags = {
  Name      = "wtp-pri_subnet"
  Environment = "production"
 }
}
```

We are not going into details of this file as we will already do it in *Terraform* later.

Next, we initialize *Terraform*:

```
tflocal init
```

It will return the following output:

```
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v4.56.0...
- Installed hashicorp/aws v4.56.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Now, we can deploy the resources by executing the following command and then answering “yes” to the prompt:

```
tflocal apply
```

The output will end like this after the prompt:

```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.
```

Enter a value: yes

```
aws_vpc.wtp_vpc: Creating...
aws_vpc.wtp_vpc: Still creating... [10s elapsed]
aws_vpc.wtp_vpc: Creation complete after 12s [id=vpc-5b45f04d]
aws_subnet.wtp_private_subnet: Creating...
aws_subnet.wtp_public_subnet: Creating...
aws_subnet.wtp_public_subnet: Creation complete after 0s [id=subnet-8a507c06]
aws_subnet.wtp_private_subnet: Still creating... [10s elapsed]
aws_subnet.wtp_private_subnet: Creation complete after 10s [id=subnet-937be6c6]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

We can check our VPCs now:

```
awslocal ec2 describe-vpcs
```

It returns:


```
{
  "Vpcs": [
    {
      "CidrBlock": "172.31.0.0/16",
      "DhcpOptionsId": "dopt-7a8b9c2d",
      "State": "available",
      "VpcId": "vpc-962aeb2a",
      "OwnerId": "000000000000",
      "InstanceTenancy": "default",
      "Ipv6CidrBlockAssociationSet": [],
      "CidrBlockAssociationSet": [
        {
          "AssociationId": "vpc-cidr-assoc-569f5d9c",
          "CidrBlock": "172.31.0.0/16",
          "CidrBlockState": {
            "State": "associated"
          }
        }
      ],
      "IsDefault": true,
      "Tags": []
    },
    {
      "CidrBlock": "10.0.0.0/16",
      "DhcpOptionsId": "dopt-7a8b9c2d",
      "State": "available",
      "VpcId": "vpc-5b45f04d",
      "OwnerId": "000000000000",
      "InstanceTenancy": "default",
      "Ipv6CidrBlockAssociationSet": [],
```

```
        "CidrBlockAssociationSet": [
          {
            "AssociationId": "vpc-cidr-assoc-0343b462",
            "CidrBlock": "10.0.0.0/16",
            "CidrBlockState": {
              "State": "associated"
            }
          }
        ],
        "IsDefault": false,
        "Tags": [
          {
            "Key": "Environment",
            "Value": "production"
          },
          {
            "Key": "Name",
            "Value": "wtp-vpc"
          }
        ]
      }
    ]
}
```


We have two VPCs in the “eu-west-1” region: the default VPC (172.31.0.0/16) and the VPC we just created (10.0.0.0/16).

Let us see the subnets:

awslocal ec2 describe-subnets  --filters "Name=vpc-id, Values=vpc-5b45f04d"

The output:


```
{
    "Subnets": [
      {
        "AvailabilityZone": "eu-west-1b",
        "AvailabilityZoneId": "euw1-az1",
        "AvailableIpAddressCount": 251,
        "CidrBlock": "10.0.2.0/24",
        "DefaultForAz": true,
        "MapPublicIpOnLaunch": false,
        "State": "available",
        "SubnetId": "subnet-937be6c6",
        "VpcId": "vpc-5b45f04d",
        "OwnerId": "000000000000",
        "AssignIpv6AddressOnCreation": false,
        "Ipv6CidrBlockAssociationSet": [],
        "Tags": [
          {
```

```
            "Key": "Environment",
            "Value": "production"
          },
          {
            "Key": "Name",
            "Value": "wtp-pri_subnet"
          }
        ],
        "SubnetArn": "arn:aws:ec2:eu-west-1:000000000000:subnet/subnet-937be6c6",
        "Ipv6Native": false,
        "PrivateDnsNameOptionsOnLaunch": {
          "HostnameType": "ip-name"
        }
      },
      {
        "AvailabilityZone": "eu-west-1a",
        "AvailabilityZoneId": "euw1-az3",
        "AvailableIpAddressCount": 251,
        "CidrBlock": "10.0.1.0/24",
        "DefaultForAz": true,
        "MapPublicIpOnLaunch": true,
        "State": "available",
        "SubnetId": "subnet-8a507c06",
        "VpcId": "vpc-5b45f04d",
        "OwnerId": "000000000000",
        "AssignIpv6AddressOnCreation": false,
        "Ipv6CidrBlockAssociationSet": [],
        "Tags": [
          {
            "Key": "Environment",
            "Value": "production"
          },
          {
            "Key": "Name",
            "Value": "wtp-pub_subnet"
          }
        ],
        "SubnetArn": "arn:aws:ec2:eu-west-1:000000000000:subnet/subnet-8a507c06",
        "Ipv6Native": false,
        "PrivateDnsNameOptionsOnLaunch": {
          "HostnameType": "ip-name"
        }
      }
    ]
}
```


We currently have one public (wtp-pub_subnet) and one private subnet (wtp-pri_subnet).

It is also possible to check out the deployed resources at https://app.localstack.cloud/. You first need to sign up for the web service. It works on the local machine, but I didn't try for the remote host.

Finally, we can terminate the stack with the following command. This termination takes place virtually. Nothing real built, nothing real destroyed!

tflocal destroy

After answering "yes" to the prompt, we get:

Plan: 0 to add, 0 to change, 3 to destroy.

Do you really want to destroy all resources?
Terraform will destroy all your managed infrastructure, as shown above.
There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

aws_subnet.wtp_public_subnet: Destroying... [id=subnet-8a507c06]
aws_subnet.wtp_private_subnet: Destroying... [id=subnet-937be6c6]
aws_subnet.wtp_private_subnet: Destruction complete after 0s
aws_subnet.wtp_public_subnet: Destruction complete after 0s
aws_vpc.wtp_vpc: Destroying... [id=vpc-5b45f04d]
aws_vpc.wtp_vpc: Destruction complete after 0s

Destroy complete! Resources: 3 destroyed.

We had created three resources before we destroyed them: One VPC and two subnets.

Since we keep the main.tf file, we can come back and rebuild everything. When exiting the container terminal, the container is still alive and ready for interactive work in the future.

sudo docker container ls -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f7c29a46267e localstack/localstack "docker-entrypoint.sh" 2 hours ago Up 2 hours (healthy) 0.0.0.0:4510-4559->4510-4559/tcp, :::4510-4559->4510-4559/tcp, 0.0.0.0:4566->4566/tcp, :::4566->4566/tcp, 5678/tcp jovial_pare

That is the end of *LocalStack*!

**Conclusion**

We wrote the code for the backbone process and organized it in a workflow with a run schedule. We used tools to inspect our code and identified and corrected the errors. Finally, we verified that the cloud infrastructure was built and ran correctly and smoothly.

Now, let us see where we are left off.

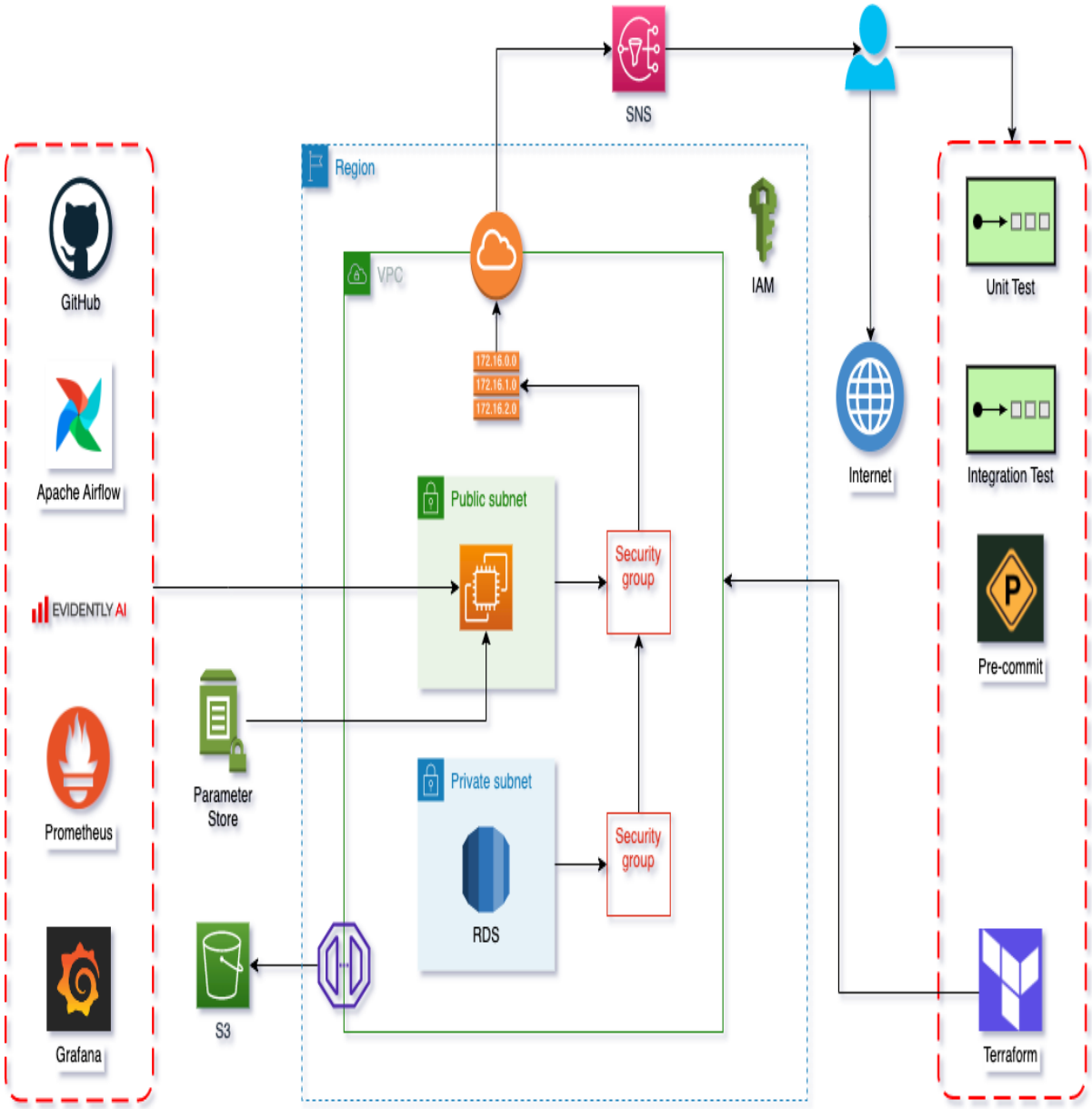



Exhibit-1: MLOps Project Diagram (Image by Author)

We finished the left part except for *GitHub Actions*. On the right side, everything other than *Terraform* is complete. The middle section is still waiting to be conquered!

The following article will show us the near-complete realization of the diagram explaining the deployment of the middle section with *Terraform*. After that, we hope to implement *GitHub Actions*, build the web interface, and finally run our application.

we have completed the testing of our code. Now, we can worry about how to deploy it.

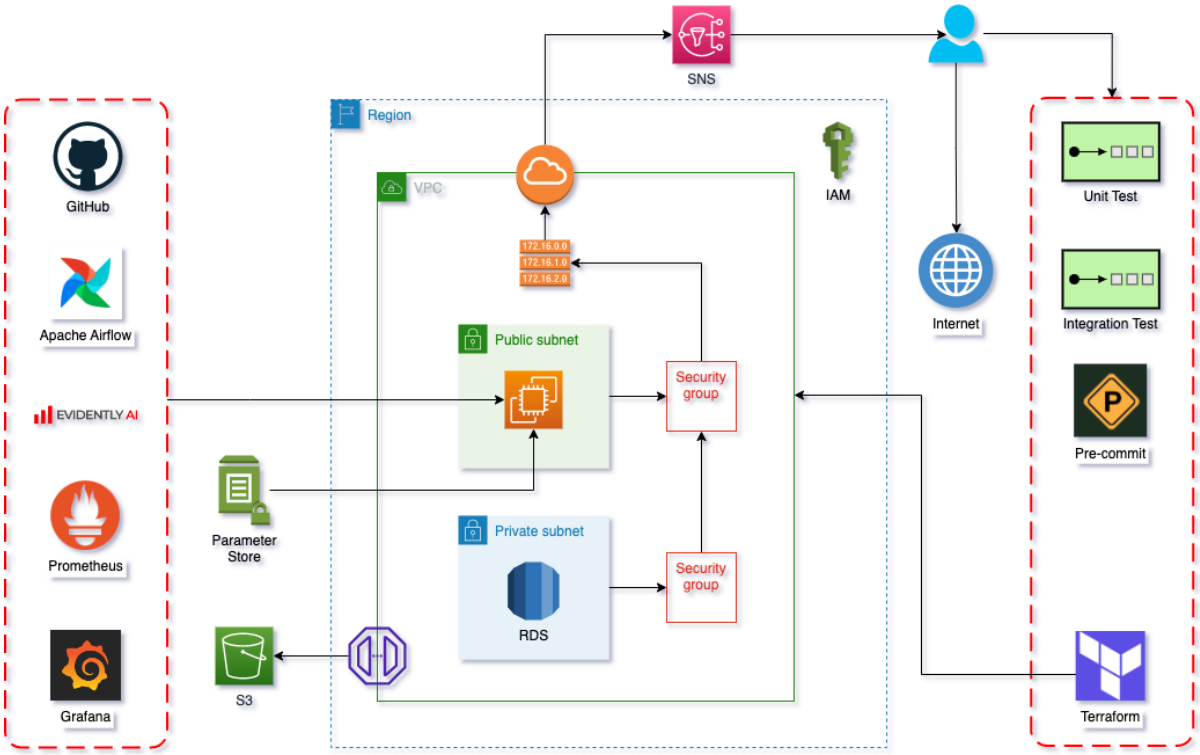



Exhibit-1: MLOps Project Diagram (Image by Author)

In this article, our task is to build all resources in the middle ground of Exhibit 1, and we will do it with *Terraform*.

**Terraform**

*HashiCorp Terraform is an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share. You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle. Terraform can manage low-level components like compute, storage, and networking resources, as well as high-level components like DNS entries and SaaS features.*

*Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs). Providers enable Terraform to work with virtually any platform or service with an accessible API.[1]*

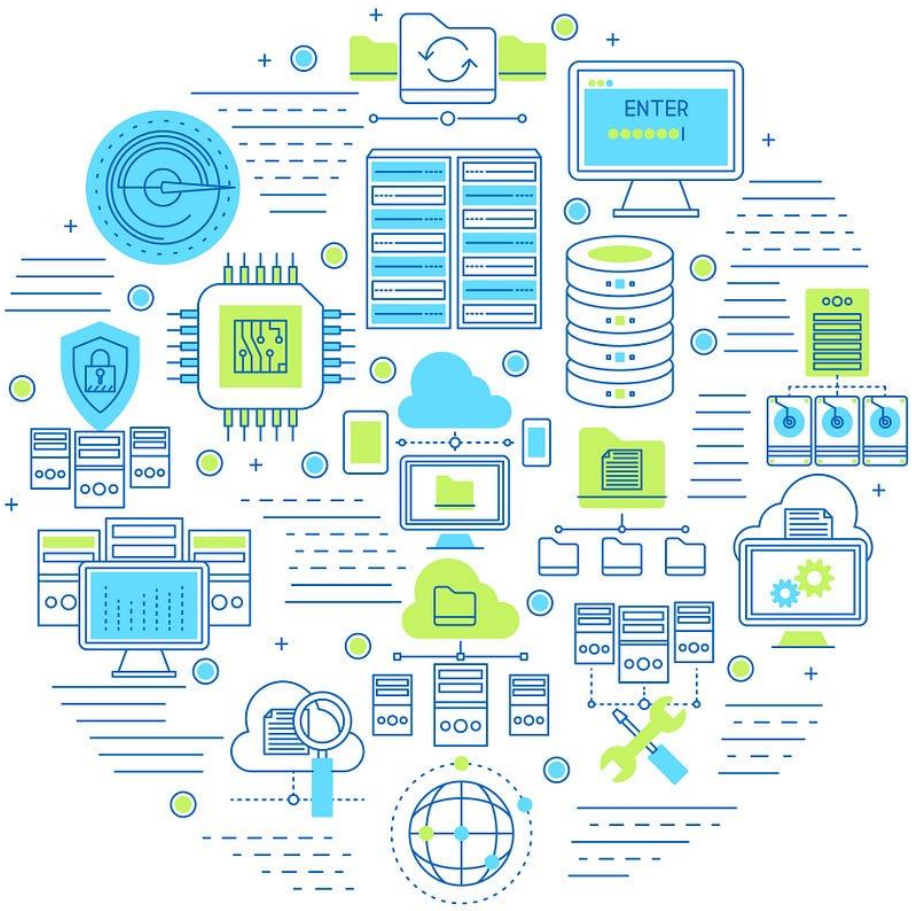



Image by macrovector on Freepik

Among those providers are Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, and DataDog.

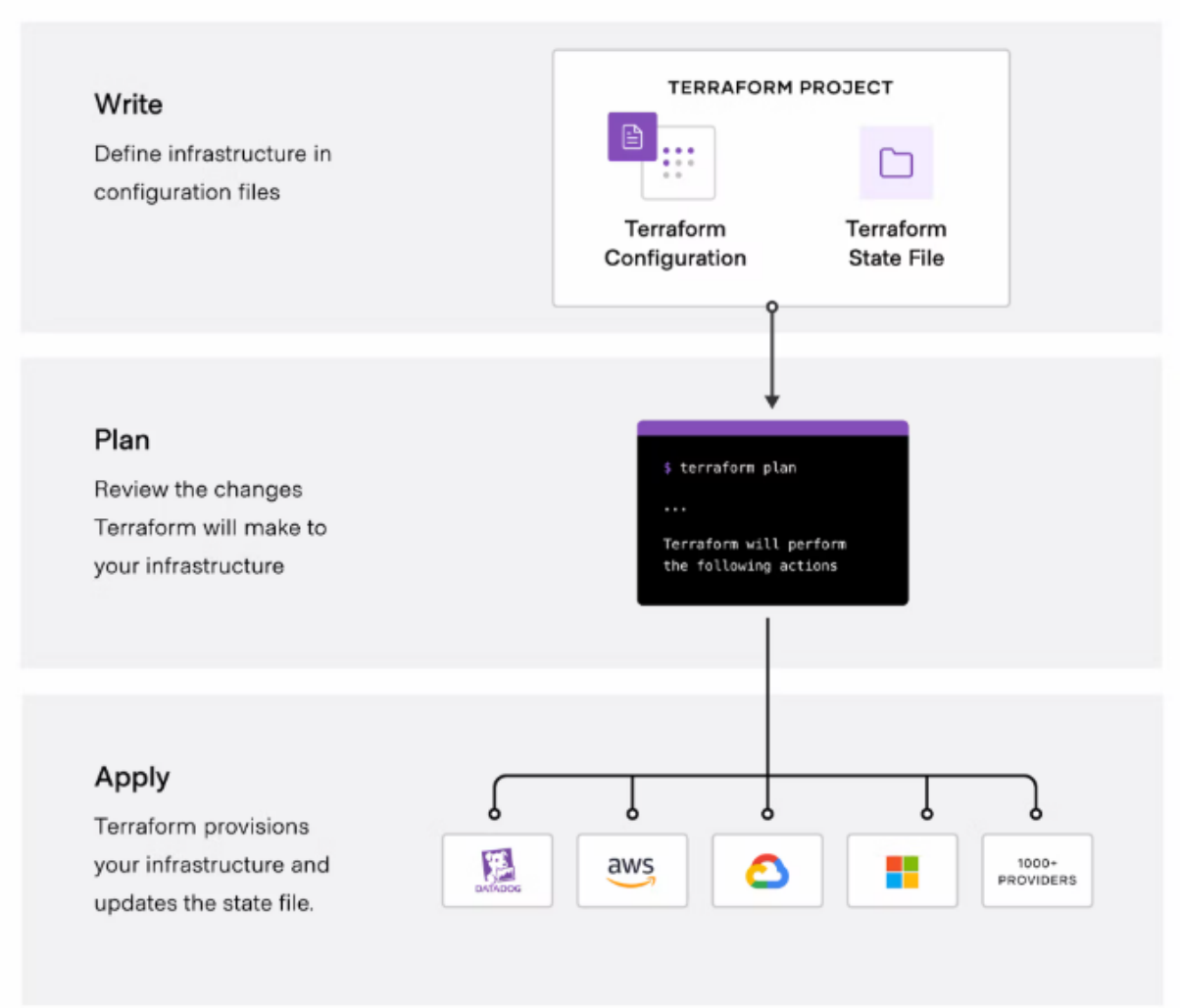



Exhibit-2: How Terraform Works? (Taken from https://developer.hashicorp.com/terraform/intro)

We build our resources, let's say, on AWS, one of the providers, through AWS' API in three stages. We first create a template where we declare and design all resources we want to deploy. However, we may create multiple configuration files that refer to each other and are activated to deploy the requested resources on the cloud. These files have .tf extensions. Some other files that have different extensions and functions also exist. We'll see them later.

In the second stage (plan), " *Terraform* creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration."[1]

In the final stage (apply), *Terraform* executes the plan to create and deploy the resources on the cloud, considering any resource dependencies. Unlike *Boto3*, no racing issue arises in *Terraform*. If we need to import a csv. file to RDS from S3; RDS must be ready before the import operation that should take place after the creation of the S3 bucket and uploading of the .csv file into the bucket. *Boto3* executes the code as a usual procedure and doesn't respect the deployment order. And that leads to stack creation errors and failure of deployment. We may end up with an S3 bucket and RDS database in which data does not exist. *Terraform* doesn't do that. It acts with the logic of "first things first."

After running apply, *Terraform* creates a file named terraform.tfstate. This file is a JSON-formatted file that stores the state of your cloud infrastructure created. If you are familiar with

finance, it is like the balance sheet, which shows the point-in-time snapshot of the financial position or assets and liabilities of a legal person. The terraform.tfstate maps our configuration to cloud resources created and keeps track of metadata. The file resides in the directory (/home/ubuntu/app/Waiter-Tips-Prediction/terraform) where *Terraform* is run. Although it is a JSON file and open to manipulation, we should avoid doing so not to confuse *Terraform*. *Terraform* updates and destroys resources based on terraform.tfstate [2]. Thus, it should be kept intact and securely and backed up.

**Installation**

In Part 7, we saw how to install *Terraform* on Ubuntu during Step III of *LocalStack* installation. *Terraform*'s page describes how to perform installations on other machines as well. Thus, we will not repeat it here. However, a more straightforward approach exists to execute from the local machine at home or work. We will use an extension on VSC to run *Terraform*. Before fulfilling the following steps, you must have an AWS account and a user.

1. Bring the Command Palette by clicking Ctrl+Shift+P (Cmd+Shift+P on Mac) in VSC, choose "Python Select Interpreter" and then Python 3.9.9 (You may have a different version).

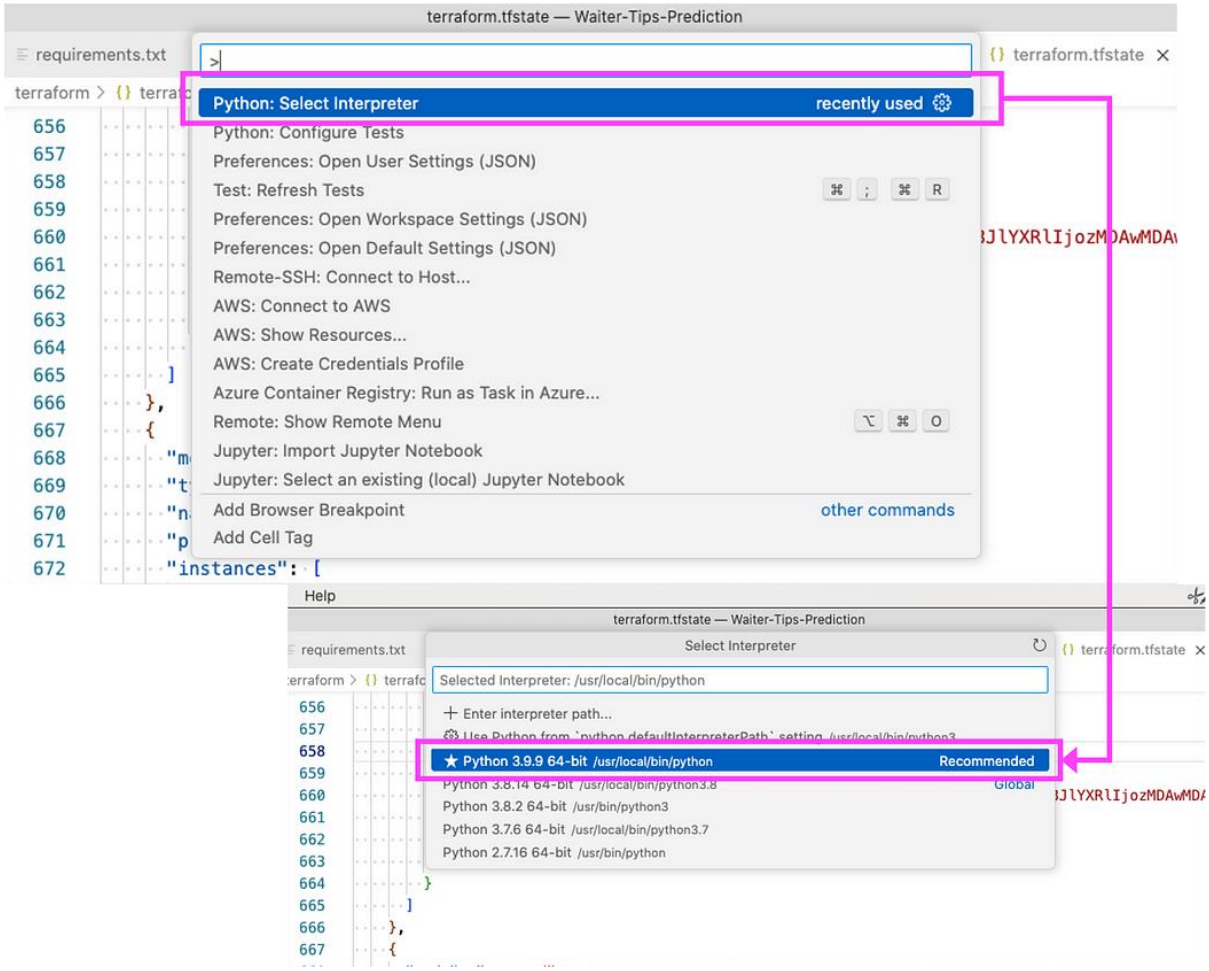



Exhibit-3: Python Interpreter (Image by Author)

2. Install the AWS Toolkit extension on VSC.

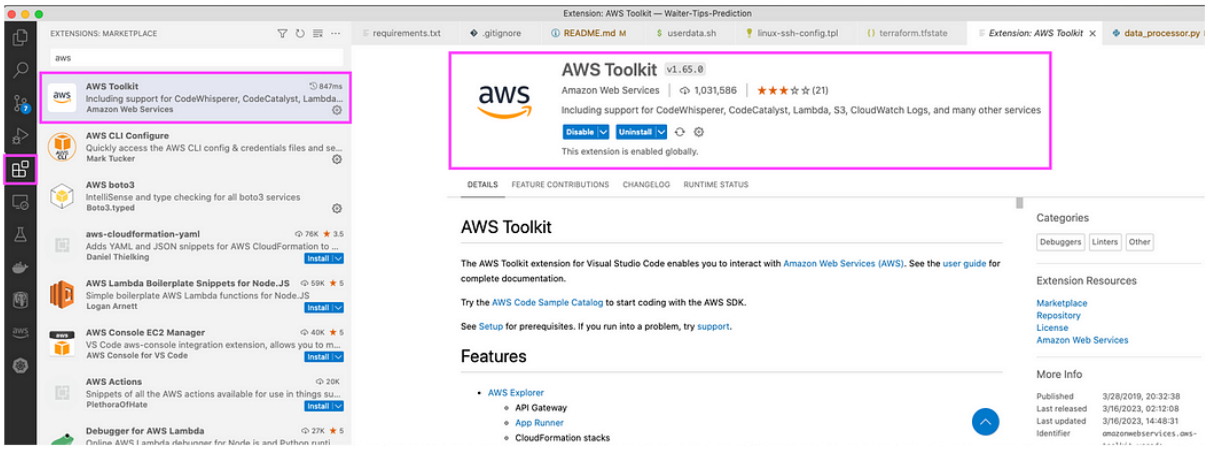



Exhibit-4: AWS Toolkit Extension on VSC (Image by Author)

3. Bring the Command Palette again by clicking Ctrl+Shift+P (Cmd+Shift+P on Mac) in VSC, and choose “AWS Create Profile.”

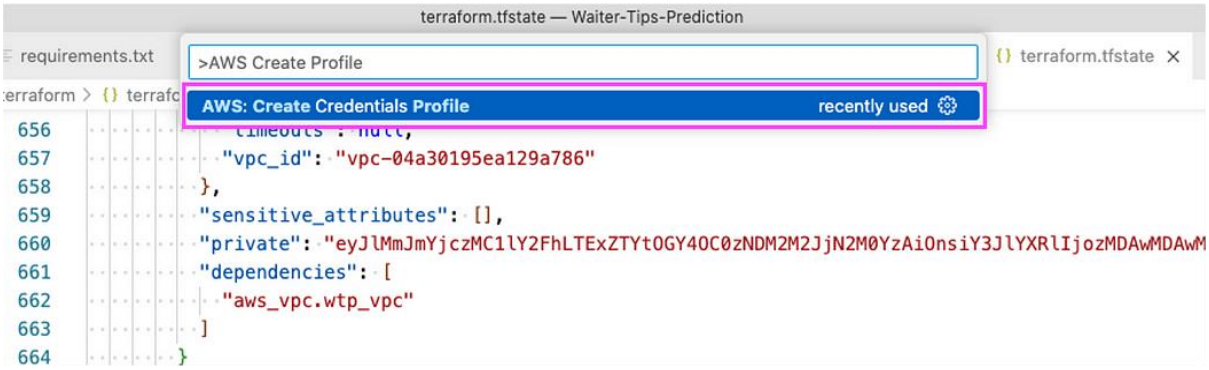



Exhibit-5: AWS Create Credentials Profile on VSC (Image by Author)

4. Enter your AWS credentials. First, type a profile name that you want to use under the AWS IAM user you created before during the AWS account opening, then type your “Access key ID” and “Secret access key.” These keys should belong to the IAM user under which you define the profile name.

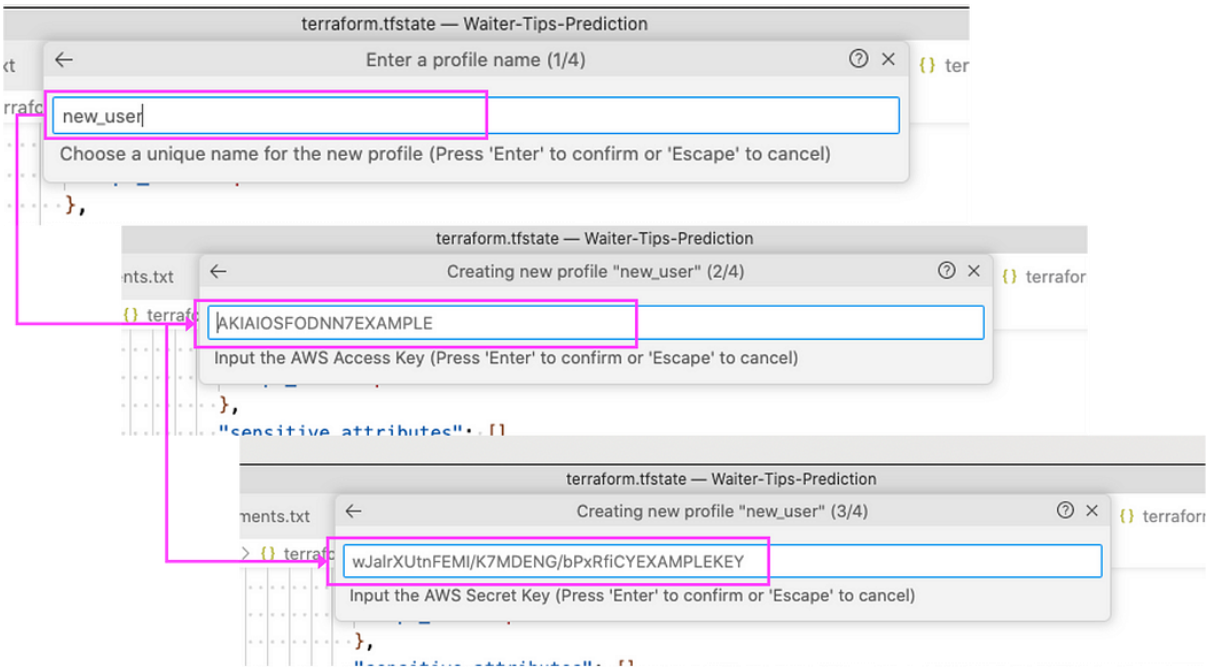



Exhibit-6: Entering AWS Credentials for the Profile on VSC (Image by Author)

We created a "named profile" (new_user) under the AWS IAM user. "A *named profile* is a collection of settings and credentials that you can apply to an AWS CLI command. When you specify a profile to run a command, the settings and credentials are used to run that command. Multiple *named profiles* can be stored in the config and credentials files."[3]

When we create an IAM user with programmatic access, a default profile is automatically created in the config and credentials files in ~/.aws directory on the local machine. Our named profile new_user in the ~/.aws/credentials file will look like this:

```
[new_user]
# This key identifies your AWS account.
aws_access_key_id = **************************
# Treat this secret key like a password. Never share it or store it in source
# control. If your secret key is ever disclosed, immediately use IAM to delete
# the key pair and create a new one.
aws_secret_access_key = ****************************************************
```

We can remove the profile by deleting the profile information from
the config and credentials files.

The default region for the created named profile is "us-east-1." If we want to change it, we can do it by editing the config file, which is located in ~/.aws/config. We append/edit the following information to the config file:

```
[profile new_user]
region = eu-west-1
output = json
```

Here, we set "eu-west-1" as the default region and the format for the output of AWS commands on the CLI as JSON for the profile new_user.

As you might guess, a “named profile” can also be directly created by appending it to the config and credentials files.

We may use a named profile by adding the --profile *profile-name* option to our CLI command. For example, to list all of our EC2 instances using the credentials and settings defined in the new_user profile, we execute the following command[3]:

aws ec2 describe-instances --profile new_user

To use a named profile for multiple commands, we can avoid specifying the profile in every command by setting the AWS_PROFILE environment variable at the command line[3]:

export AWS_PROFILE=new_user

All this hassle is due to our willingness to run *Terraform* operations under a new profile name. If we are happy with the default profile, we may skip the third and fourth steps.

5. Install the *Terraform* extension by Anton Kulikov on VSC.

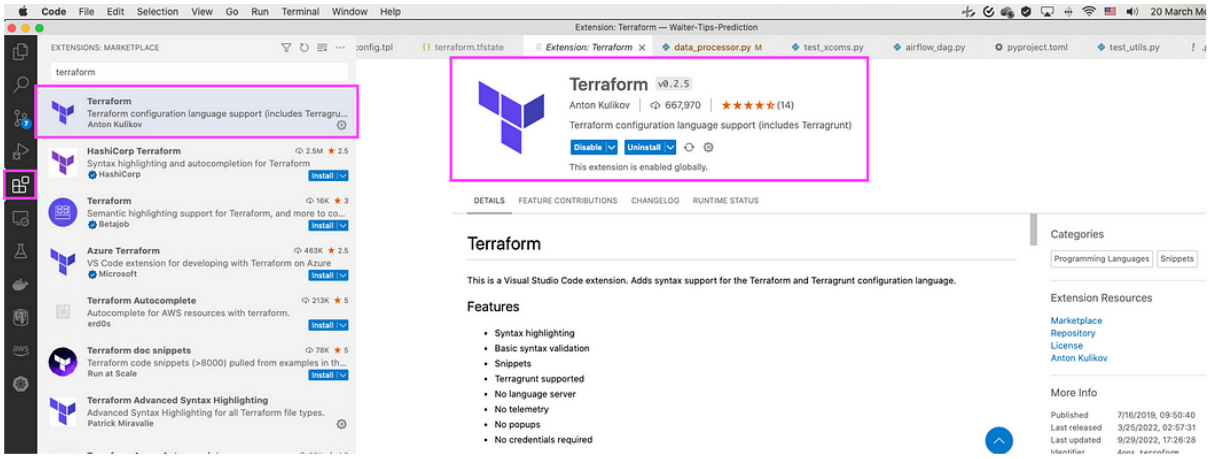



Exhibit-7: Installing Terraform Extension on VSC (Image by Author)

6. Create a new directory (“terraform”) under the project directory if we haven’t done it before. The project directory is /home/ubuntu/app/Waiter-Tips-Prediction. This directory may have a different address path on your local machine.

The installation step is complete. We can continue with the configuration.

**Configuration — Write Stage**

In this step, we will create configuration files in the “terraform” folder. The configuration phase is the first step in the “write-plan-apply” process we saw earlier. While it is possible to configure everything in one file, using multiple files is a good practice.

*providers.tf*

The first file we are going to generate is providers.tf:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
```

```
 }
}

provider "aws" {
 region                  = "eu-west-1"
 shared_config_files     = ["~/.aws/config"]
 shared_credentials_files = ["~/.aws/credentials"]
 profile                 = "new_user"
}
```

We set AWS as the provider, which means this cloud platform will host all our resources. “The provider block configures the specified provider, in this case aws. A provider is a plugin that *Terraform* uses to create and manage your resources.”[4] The information in the provider block is already familiar, as we created and edited them during the installation phase.

We can use multiple provider blocks to manage resources from different providers. For example, we could pass the IP address of our AWS EC2 instance to a monitoring resource from DataDog[4].

*main.tf*

The second file we need to create is main.tf, where we will declare our resources to be deployed on AWS. We may create different files for different resources as long as they reside in the same directory. For example, storage services can be configured in one .tf file, while compute resources can be defined in another. For this project, the main.tf seems to be sufficient. The content of the main.tf file is relatively long, so we’ll see a few blocks here to understand how it works. You can find the main.tf file in full length here:

**Machine-Learning/main.tf at main · hsaltan/Machine-Learning**

This is about machine learning projects. Contribute to hsaltan/Machine-Learning development by creating an account on...

github.com

Let us see the first block:

```
resource "aws_iam_policy" "wtp_ec2_policy" {
 name   = "wtp_ec2_policy"
 path   = "/"
 policy = data.aws_iam_policy_document.wtp_ec2_policy_doc.json
}
```

We use the resource block to define the resources to deploy on the cloud. “A resource might be a physical or virtual component such as an EC2 instance, or it can be a logical resource such as a Heroku application.”[4]

The resource block defines the component with two strings: the “resource type” and the “resource name.” In the above snippet, the resource type is aws_iam_policy. It

tells *Terraform* that we need to create an IAM policy on AWS. Resource types are built-in names and start with the prefix of the cloud provider, which, in our case, is aws. Resource type is followed by the user-defined resource name. In the above example, our IAM policy name is wtp_ec2_policy.

The resource type and resource name together make up a unique ID for the resource. Thus, aws_iam_policy.wtp_ec2_policy is the ID of our IAM policy.

Depending on the resource type, each resource block contains several arguments, such as vpc_id, destination_cidr_block, block_public_acls, bucket, etc. Some of these arguments are required, and some are optional. If you are familiar with *Boto3*, you should not have difficulty understanding "arguments." Even if you do not know about *Boto3*, *Terraform* has excellent documentation that explains the resources and arguments.

As one of the arguments in the example above, the policy receives its value from the data block in another file. Let us leave it aside for now and keep going along the main.tf file.

Once we formed the IAM policy, the next is to create the IAM role:

```
resource "aws_iam_role" "wtp_ec2_role" {
 name = "wtp_ec2_role"

 assume_role_policy = jsonencode({
  Version = "2012-10-17"
  Statement = [
   {
    Action = "sts:AssumeRole"
    Effect = "Allow"
    Sid    = ""
    Principal = {
     Service = "ec2.amazonaws.com"
    }
   },
  ]
 })

 tags = {
  tag-key = "wtp_ec2_role"
 }
}
```

In the resource block, we see the resource type and name: aws_iam_role , wtp_ec2_role. An IAM role is an IAM identity that has specific permissions.

In this example, the assume_role_policy argument, as in the case when creating an IAM role on the AWS console, allows an EC2 instance to assume the role using the AssumeRole operation and use the permissions attached to this role. Permissions are defined in the IAM policy we generated above. Lastly, we set a tag (wtp_ec2_role), which is optional.

By now, we have an IAM role and policy. Next, we need to attach the policy to the role:

```
resource "aws_iam_role_policy_attachment" "wtp_ec2_policy_attach" {
  role       = aws_iam_role.wtp_ec2_role.name
  policy_arn = aws_iam_policy.wtp_ec2_policy.arn
}
```

Things are getting easier as we evolve along the configuration. We create an aws_iam_role_policy_attachment resource, which is named wtp_ec2_policy_attach. We tell *Terraform* to attach the policy to the role. In this setting, the role argument needs the role's name, and the policy_arn needs the policy's arn. As mentioned, we uniquely define resources by their types and names together. The ID for the role is aws_iam_role.wtp_ec2_role , and the ID for the policy is aws_iam_policy.wtp_ec2_policy. The required name and arn properties are then derived from the corresponding objects defined by the IDs.

After the attachment is made, we need to create an instance profile for EC2 for it to assume the role defined:

```
resource "aws_iam_instance_profile" "wtp_ec2_profile" {
  name = "wtp_ec2_profile"
  role = aws_iam_role.wtp_ec2_role.name
}
```

"An instance profile is a container for an IAM role that you can use to pass role information to an EC2 instance when the instance starts."[5]

The instance profile needs the role name, which already exists. What we configure in the *Terraform* configuration files exactly matches what we should do when working on the cloud console. There is nothing extra.

The next step is to get the key pair for our EC2. For this, we need to create a key pair first. On the terminal opened in the "terraform" directory, we execute the following command first:

```
ssh-keygen -t ed25519
```

This key is more secure than the typical RSA key. The prompt will ask us the file in which to save the key. On Mac, public keys are located in the ~/.ssh directory. So, we will keep our key in .ssh directory after renaming it as "wtpkey":

```
/dir_1/dir_2/.ssh/wtpkey
```

/dir_1/dir_2/ are just placeholders to exemplify the path to the .ssh folder on the local machine. Then we press "Enter" twice to finalize the creation of our key. In the .ssh directory, two files will appear: wtpkey, the private key, and wtpkey.pub, the public key.

While creating the key pair resource on *Terraform*, it's possible to define the public key argument by directly passing the content of wtpkey.pub file, but this will look messy. A more elegant way is to refer to the public key file as shown below:

```
resource "aws_key_pair" "wtp_key" {
  key_name   = "wtpkey"
  public_key = file("~/.ssh/wtpkey.pub")
}
```

The resource for our key is identified as aws_key_pair.wtp_key.

The key_name is the name of the key pair. If neither key_name nor key_name_prefix is provided, Terraform will create a unique key name using the prefix terraform-.

The file function reads the contents of our file (wtpkey.pub) at the path we gave (~/.ssh/wtpkey.pub) and returns them as a string.

Now, we can configure our EC2 server:

```
resource "aws_instance" "wtp_node" {
 ami                  = data.aws_ami.server_ami.id
 instance_type        = "t2.large"
 key_name             = aws_key_pair.wtp_key.id
 vpc_security_group_ids = [aws_security_group.wtp_ec2_sg.id]
 subnet_id            = aws_subnet.wtp_public_subnet.id
 iam_instance_profile   = aws_iam_instance_profile.wtp_ec2_profile.name
 user_data            = file("userdata.sh")

 root_block_device {
  volume_size = 30
 }

 tags = {
  Name      = "wtp-ec2"
  Environment = "production"
 }

 provisioner "local-exec" {
  command = templatefile("linux-ssh-config.tpl", {
   hostname     = self.public_ip,
   user       = "ubuntu",
   identityfile = "~/.ssh/wtpkey"
  })
  interpreter = ["bash", "-c"]
 }
}
```

The resource block names aws_instance as wtp_node, and retrieves the arguments key_name, and iam_instance_profile from the configuration blocks we did above. We didn't explain vpc_security_group_ids and subnet_id here, but they are defined in the main.tf file in a similar manner. We set the instance_type, volume_size, and tags explicitly.

What we see new above is the provisioner block and userdata.sh file. We have encountered the data. object before. We divert our attention from main.tf to these new components for a time.

*linux-ssh-config.tpl*

The .ssh directory is the default location for all configuration and authentication files that enable users to connect to a server. The config file in the directory lists the information about the server and looks like this:

```
Host wtp
  HostName 3.249.105.159
  User ubuntu
  IdentityFile ~/.ssh/wtpkey
```

Host is the section that contains the connection details under the name wtp. Any name is okay, though. When we type ssh wtp on the bash terminal, the ssh client will read and use these details to enable connection to the server.

HostName is the IP address of the server and User is the username. IdentityFile configures the location of identity keys in the OpenSSH client configuration files, usually /etc/ssh/ssh_config or .ssh/config in the user's home directory. "An identity key is a private key that is used in SSH for granting access to servers. They are a kind of SSH key, used for public key authentication. The default location for identity keys on Unix/Linux systems is the .ssh directory in each user's home directory."[6]

The linux-ssh-config.tpl is a user-defined file that contains the template for the connection details to be stored in the config file in the .ssh directory. The .tpl extension is an abbreviation for "template" and a label for template files. The file is like this:

```
cat << EOF >> ~/.ssh/config

Host ${hostname}
  HostName ${hostname}
  User ${user}
  IdentityFile ${identityfile}
EOF
```

The first line (cat << EOF >> ~/.ssh/config) adds the following multi-line string to ~/.ssh/config file in bash. If you have multiple variables to write to ~/.ssh/config as in our case, use the above HereDoc format, which records everything between the lines beginning with cat and EOF to the configuration file.

"A here document (**HereDoc**) is a section of code that acts as a separate file. A HereDoc is a multiline string or a file literal for sending input streams to other commands and programs."[7]

The variables in the HereDoc will receive their values from another document to pass it to ~/.ssh/config , and we'll see that later. The linux-ssh-config.tpl is a template file for Linux and Mac. A slightly different content exists for Windows.

In the EC2 instance configuration of the main.tf file, we used the provisioner block. Provisioners are used for modeling specific actions on the local machine or a remote host to prepare servers or other infrastructure objects for service. Provisioners don't affect the state, which means we will see no effect in the "plan" and terraform.tfstate file. *Terraform* recommends using other options instead of provisioners and provisioners only as the last resort option[8]. However, for this project, we can use it.

The local-exec provisioner invokes a local executable after the creation of a resource. The choice of local-exec invokes a process on the machine which runs *Terraform*, not on the resource (EC2 in the example)[8].

The command argument describes the command that will be run on the bash terminal. This execution occurs with the help of the interpreter argument. In the interpreter argument, -c refers to the command, and bash indicates where this command will be executed.

In the command argument, the templatefile function reads the file linux-ssh-config.tpl and renders its content as a template using a supplied set of template variables (hostname, user, identityfile)[9]. When the command is executed on the bash terminal, the templatefile function will replace the variables in the linux-ssh-config.tpl template file with the values (hostname, user, identityfile) specified in the provisioner block. Sequentially, the linux-ssh-config.tpl will save those variables in ~/.ssh/config.

In the templatefile, hostname is self.public_ip , where self refers to the resource, aws_instance.wtp_node. It takes the public IP address of the EC2 instance. As we saw in the config file in the .ssh directory, HostName was the server IP address.

Last step to make the linux-ssh-config.tpl file operational is to install the Remote- SSH extension on VSC.

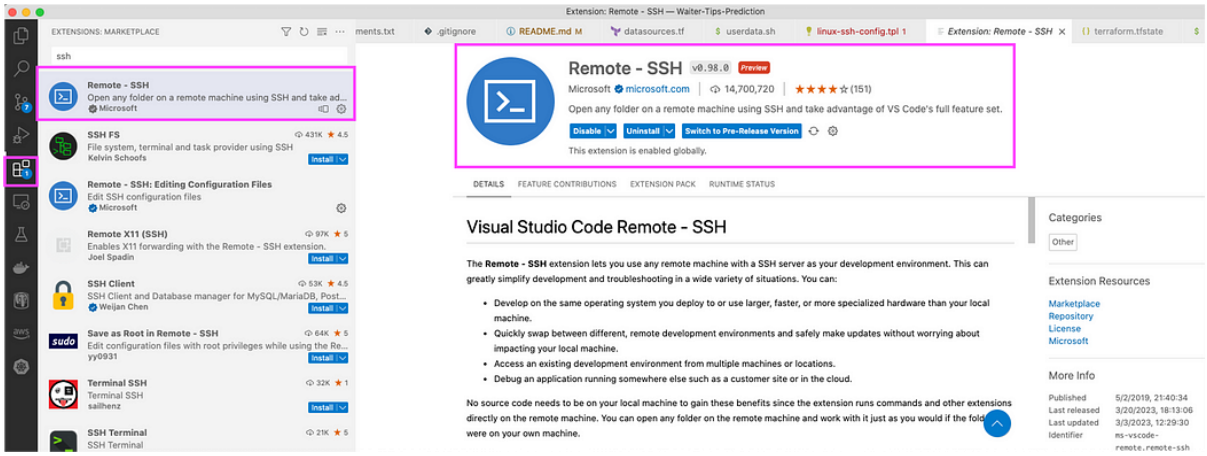



Exhibit-8: Installing Remote — SSH Extension on VSC (Image by Author)

Now, we'll find out from which planet the data. object comes from!

*datasources.tf*

In the "terraform" folder, we create another file named datasources.tf. "*Data sources* allow *Terraform* to use the information defined outside of *Terraform*, defined by another separate *Terraform* configuration, or modified by functions. A data source is accessed via a special kind of resource known as a *data resource*, declared using a data block."[10]

The first data block in the datasources.tf file is:

```
data "aws_ami" "server_ami" {
 most_recent = true

 owners = ["***************"]

 filter {
  name   = "name"
  values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
 }
```

```
 filter {
  name   = "virtualization-type"
  values = ["hvm"]
 }
}
```

The data block asks *Terraform* to read from a given data source aws_ami, and return the result under the local name “server_ami.” The name is defined at the module level and used for referring to the resource only within the scope of the module. It has no operational effect outside the scope of the module[10].

The type (aws_ami) and the name (server_ami) in the data block compose a unique identifier within the module and point to a given resource, a data instance.

Each data source has a single object (such as aws_ami). Curly brackets in the data block define the data source-specific arguments. Therefore, depending on the source, arguments can vary. In our example, aws_ami as a source has three arguments such as most_recent, owners , and filter.

In our data block, we request *Terraform* to create a data instance for EC2 AMI under the name server_ami and constrain the AMI options such that:

1. They will be owned by users specified in the list (Currently, we have only one, but it can be many. Replace “*” with your AWS account number.)
2. The most recent version of AMIs that match ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*” will be chosen. We can see this identifier in the “AMI name” column on the AWS EC2 console.
3. AMI will have hvm as the virtualization type.

Now, if we go back to the aws_instance resource block in the main.tf file and look at the upper part:

```
resource "aws_instance" "wtp_node" {
 ami                  = data.aws_ami.server_ami.id
 instance_type        = "t2.large"
 key_name             = aws_key_pair.wtp_key.id
 vpc_security_group_ids = [aws_security_group.wtp_ec2_sg.id]
 subnet_id            = aws_subnet.wtp_public_subnet.id
 iam_instance_profile   = aws_iam_instance_profile.wtp_ec2_profile.name
 user_data            = file("userdata.sh")
```

we clearly see data.aws_ami.server_ami.id. This identifier refers to the AMI id of the data source aws_ami , which goes under the name server_ami , and is defined in the data block created in the module contained in the datasources.tf file! It’s only AMI ID, for God’s sake!

In the AWS policy document we saw earlier,

```
resource "aws_iam_policy" "wtp_ec2_policy" {
 name   = "wtp_ec2_policy"
 path   = "/"
```

```
  policy = data.aws_iam_policy_document.wtp_ec2_policy_doc.json
}
```

the policy argument referred to another data block, which was:

```
data "aws_iam_policy_document" "wtp_ec2_policy_doc" {
  statement {

    sid = "1"

    effect = "Allow"

    actions = [
      "rds:*",
      "s3:*",
      "sns:*",
      "ssm:*"
    ]

    resources = ["*"]
  }
}
```

When we create a “policy” on the AWS console, we need to define a “policy document” that sets the actions authorized or unauthorized and the resources that those actions will affect. Therefore, a “policy” possesses a “policy document” in AWS.

In *Terraform*, we define the “policy document” in the data block in the datasources.tf file and create the IAM “policy” in the main.tf . The “policy” imports the required permissions from the datasources.tf file through the policy argument.

As “policy documents” are in JSON format in AWS, we should insert “policy document” in this format also. Thus, we need to use the .json extension in the policy argument of the resource aws_iam_policy.

As a result, in our example, we gave Superman powers to our EC2 instance! It is not a good practice, but we can live with that for our application.

We use another file for the user data in the aws_instance resource in main.tf and set it as file(“userdata.sh”). Now, let us look at this new file.

*userdata.sh*

We use user data in EC2 instances to bootstrap them. Bootstrapping means passing user data to the instance during launch. This data contains standard configuration tasks that will be performed automatically during the startup. The data is inserted into the user data section on the configuration page of the EC2 and is limited to 16KB.

We could have inserted the user data as the code block in user_data argument of the aws_instance resource, but we preferred to pass data through a .sh file. That is a neater way. The .sh file is a shell script used for installing an application or performing tasks in Linux. So,

our userdata.sh file will execute all written tasks when the EC2 instance starts. That is how we plan in *Terraform*. userdata.sh can be found here:

**Machine-Learning/userdata.sh at main · hsaltan/Machine-Learning**

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

github.com

This configuration is not ideal because we didn't use containers in the application. In a typical containerized application, docker-compose handles all installation and configuration tasks. That is something we plan to do in another project. We can see and explain the code in the userdata.sh file step-by-step as follows:

1. Update each outdated package and dependency in Ubuntu.

```
#!/bin/bash
sudo apt update -y
```

2. Change the current directory to /home/ubuntu and install *Python* 3.9.

```
cd /home/ubuntu
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa -y
sudo apt install python3.9 -y
```

3. Set an alias for *Python* 3.9. Any time we type python3, the execution will take place under python3.9. The following command only holds for the current session.

```
alias python3=python3.9
```

4. Install AWS CLI.

```
sudo apt install awscli -y
```

5. Create a directory named app. Change the current directory to app. Create another directory there, which is called Waiter-Tips-Prediction. Change back the current directory to the parent directory.

```
mkdir app
cd ./app
mkdir Waiter-Tips-Prediction
cd ..
```

6. Grant Read, Write, and Execute permissions in all folders and files in the /home/ubuntu/app directory for users and others. Don't give this permission to the /home/ubuntu directory as it spoils the authorized_key in .ssh folder on EC2.

```
sudo chmod -R 707 /home/ubuntu/app
```

7. Add the project directory to the path and append the path to the .bashrc file. The .bashrc file is a script file executed when a user logs in. It contains the necessary setting up or enabling configurations for the terminal session.

Append the alias command to .bashrc

Assign the *PostgreSQL* password to the environment variable and append the whole statement to .bashrc. We'll discuss *PostgreSQL* in the following article, but we should set the password now.

Save and activate the changes.

```
echo 'export PATH="${HOME}/app/Waiter-Tips-Prediction:${PATH}"' >> .bashrc
echo 'alias python3=python3.9' >> .bashrc
echo 'export PGPASSWORD=password' >> .bashrc
source .bashrc
```

8. Install pip package manager.

```
sudo apt install python3-pip -y
```

9. Copy the requirements.txt, prometheus-config.yml, and start.sh files from the S3 bucket to the project directory.

```
aws s3 cp s3://s3b-tip-predictor/config/requirements.txt ./app/Waiter-Tips-Prediction
aws s3 cp s3://s3b-tip-predictor/config/prometheus-config.yml ./app/Waiter-Tips-Prediction
aws s3 cp s3://s3b-tip-predictor/config/start.sh ./app/Waiter-Tips-Prediction
```

10. Allow executable permissions to start.sh file.

```
chmod +x ./app/Waiter-Tips-Prediction/start.sh
```

11. Change the current directory to ./app/Waiter-Tips-Prediction.

```
cd ./app/Waiter-Tips-Prediction
```

12. Install Flask and other libraries in the requirements.txt file.

```
python3.9 -m pip install flask
pip install -r requirements.txt
```

13. Execute the following command to install build tools. They are necessary to install some packages with Homebrew.

```
sudo apt-get install build-essential procps curl file git -y
```

14. Install *Grafana* as described on their page.

```
sudo apt-get install -y apt-transport-https
sudo apt-get install -y software-properties-common wget
sudo wget -q -O /usr/share/keyrings/grafana.key https://packages.grafana.com/gpg.key
echo "deb [signed-by=/usr/share/keyrings/grafana.key] https://packages.grafana.com/oss/deb
stable main" | sudo tee -a /etc/apt/sources.list.d/grafana.list
sudo apt-get update
sudo apt-get install grafana -y
```

15. Start the *Grafana* service.

```
sudo systemctl daemon-reload
sudo systemctl start grafana-server
```

16. Configure the *Grafana* server to start at boot.

```
sudo systemctl enable grafana-server.service
```

17. Install *PostgreSQL*.

```
sudo apt install -y postgresql postgresql-contrib
```

18. Start *PostgreSQL*.

```
sudo systemctl start postgresql.service
```

19. Change the current directory to two levels up.

```
cd ../../
```

20. Move the *Airflow* home directory to the current directory. The below command will collect all *Airflow* files within the “Waiter-Tips-Prediction” folder when we run the DAG. By saving and activating this command, we automate the operation.

```
echo 'export AIRFLOW_HOME=/home/ubuntu/app/Waiter-Tips-Prediction' >> .bashrc
source .bashrc
```

That’s all with userdata.sh! When we launch the EC2 instance, it will execute all the above lines in order.

Now, we can return to the main.tf file, and continue from where we are left off. We have finished the EC2 deployment in the plan and can move on to starting another resource. It’s the S3 bucket:

```
resource "aws_s3_bucket" "s3b-tip-predictor" {
 bucket = var.bucket
 tags = {
  Name       = "my-bucket"
  Environment = "production"
 }
}
```

The resource block introduces the S3 bucket with the type aws_s3_bucket . The resource has the id “aws_s3_bucket.s3b-tip-predictor” and tags. Here, we notice another object var. As you may guess, we generate a new document to which the var object refers. And that is the variables.tf file.

*variables.tf*

During the execution of the code, it may become more practical to input particular values during the runtime without hardcoding, which will provide flexibility to the user.

To launch variables for a dynamic environment, we create a file named variables.tf and declare variables in this document. variables.tf will allow users to pass values on CLI for the resources

during the runtime when destroy, plan, and apply commands are used. The below piece of code defines the variable block used for our S3 bucket:

```
variable "bucket" {
  type        = string
  description = "Bucket name"
  default     = "s3b-tip-predictor"
}
```

The variable keyword is followed by the label, which is the user-given name of the variable. All variables in the file must be unique. The label helps assign a value to the variable from outside and refers to the variable's value from within the module[11].

In the variable block, we see three arguments even though they can be more. They are[11]:

- type: The type argument specifies what value types are acceptable for the variable. When a type constraint does not exist, a value of any type is admissible. *Terraform* recommends specifying the type as it will serve as a helpful reminder for users and delivery of error messages if the wrong input type is in place. In the example, string type is acceptable.
- description: This argument specifies the input variable's documentation. It enables us to describe the purpose of the variable. The description should concisely explain the purpose of the variable and what kind of value is expected, and be written from the user's perspective as it will appear on the screen during runtime.
- default: This argument makes the variable optional. The default value is used if no value is set by the user when calling the module or running *Terraform*.

Our variable block asks the user to give a string name for the S3 bucket, which is prompted under theBucket name description on CLI, and in case the user sets no value during runtime, the name for the bucket will be s3b-tip-predictor.

We specify five variables for user input in the variables.tf file, which you can find here:

**Machine-Learning/variables.tf at main · hsaltan/Machine-Learning**

This is about machine learning projects. Contribute to hsaltan/Machine-Learning development by creating an account on...

github.com

Eventually, our resource aws_s3_bucket in the main.tf file will take the bucket name from the user on CLI by its bucket argument, which refers to the "bucket" variable in the variables.tf file! Other variables follow the same pattern. They receive their inputs from users on CLI during runtime and are passed into the arguments of resources in the main file.

Creating the variables.tf file is one of the three alternatives to define variables. A second alternative is forming the terraform.tfvars file. For example, we could have described the bucket variable in terraform.tfvars as follows:

bucket = "s3b-tip-predictor"

The terraform.tfvars file gives the default names for the definition of variables, taking precedence over the default value in the variable.tf file. In other words, terraform.tfvars allows us to define all variables before “plan and apply” instead of doing it during runtime. When several variables exist, this may be the efficient way. Then, we specify that file on the command line with -var-file:

terraform apply -var-file="terraform.tfvars"

A third alternative to defining variables is through the command line. Executing the following commands on the terminal takes precedence over the previous two options (terraform.tfvars, variables.tf):

terraform plan -var="bucket=s3b-tip-predictor"
terraform apply -var="bucket=s3b-tip-predictor"

The third alternative is probably more suitable when one or two variables exist.

We can integrate conditionals, which can be necessary to provide users with options, such as the operating system used, cloud provider, etc. However, it is beyond the scope of this article.

Now, let us see how we can upload an object to this bucket:

```
resource "aws_s3_object" "object1" {
  bucket = aws_s3_bucket.s3b-tip-predictor.id
  acl    = "private"
  key    = "config/requirements.txt"
  source = "../requirements.txt"
}
```

Our resource is aws_s3_object, and its name is object1. We refer to our unique bucket as aws_s3_bucket.s3b-tip-predictor. We need its ID for the “put object” operation. We set the S3 access control list (ACL) as private. The upload object (requirements.txt) is in the parent directory since the main.tf file resides in the “terraform” folder. The upload file is where the “terraform” folder resides. We upload this object to the config folder in our s3b-tip-predictor bucket as requirements.txt.

We configure all other resources in the main file similarly. *Terraform* documentation describes and explains everything and gives example codes.

*outputs.tf*

The outputs.tf is another document we want to create. We may need to output values about our infrastructure for informational purposes. This information may be helpful to use for other *Terraform* configurations. Output values perform the same function as return values in *Python*.

We have one output declaration in the outputs.tf file:

```
# Generate output of IAM role name
output "iam_instance_profile_name" {
  value      = aws_iam_instance_profile.wtp_ec2_profile.name
```

```
  description = "IAM role name"
}
```

The output block declares a single output value, which is then exported by a module. “The label immediately after the output keyword is the name (iam_instance_profile_name), which must be a valid identifier. In a root module, this name is displayed to the user; in a child module, it can be used to access the output's value.”[12]

The value argument takes an expression, and its result is returned to the user on CLI. In this example, the expression is the name of the aws_iam_instance_profile resource that goes under the label wtp_ec2_profile , which we created earlier. The output is then wtp_ec2_profile (see above).

The description argument is optional and used to describe the purpose of output value. This argument should concisely explain the purpose and the value type expected, and it should be written from the user's perspective, not from the perspective of the module maintainer[12].

*Terraform* apply command will add the output to the state (terraform.tfstate).

*terraformstate.tf*

The terraformstate.tf file is created by *Terraform* during the “apply” stage and lists all resources built. It is quite a long file and states all configuration details. A tiny part of it looks like this:


```
{
 "version": 4,
 "terraform_version": "1.2.9",
 "serial": 101,
 "lineage": "c47fd761-ca00-d370-91e8-2e8b9fa3dd15",
 "outputs": {
  "iam_instance_profile_name": {
   "value": "wtp_ec2_profile",
   "type": "string"
  }
 },
 "resources": [
  {
   "mode": "data",
   "type": "aws_ami",
   "name": "server_ami",
   "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
   "instances": [
    {
     "schema_version": 0,
     "attributes": {
      "architecture": "x86_64",
      "arn": "arn:aws:ec2:eu-west-1::image/ami-04e2e94de097d3986",
      "block_device_mappings": [
       {
        "device_name": "/dev/sda1",
        "ebs": {
```

```
            "delete_on_termination": "true",
            "encrypted": "false",
            "iops": "0",
            "snapshot_id": "snap-0d3fb932572618072",
            "throughput": "0",
            "volume_size": "8",
            "volume_type": "gp2"
          },
          "no_device": "",
          "virtual_name": ""
        },
        {
          "device_name": "/dev/sdb",
          "ebs": {},
          "no_device": "",
          "virtual_name": "ephemeral0"
        },
        {
          "device_name": "/dev/sdc",
          "ebs": {},
          "no_device": "",
          "virtual_name": "ephemeral1"
        }
      ],
```


...

Terraform uses the “state” recorded during the construction phase when the destruction of the infrastructure occurs.

All the files we have created are stored in the “terraform” folder, including terraformstate.tf.

We have completed the configuration or “write” stage. Now, it’s time to build and deploy our resources on AWS.

**Plan and Apply Stages**

In this section, we’ll see the CLI commands to plan and deploy our resources on the cloud with *Terraform*. All these commands are executed on the terminal opened within the “terraform” folder.

First, initialize *Terraform*:

terraform init

Correct the format of files:

terraform fmt

The above command will correct and standardize the format of the code in the files, such as spacing, aligning the lines, etc.

Then plan:

terraform plan

The plan command lists the resources to be created on the screen. Sequentially, apply:

terraform apply

The apply command will create all resources after typing yes to confirm the operations. If you want to build resources without the confirmation prompt, then execute:

terraform apply -auto-approve

After this, other commands exist to get more information about the infrastructure. Let us see some of them.

List AWS resources deployed:

terraform state list

See the details of a resource (aws_instance.wtp_node):

terraform state show aws_instance.wtp_node

Check the entire state (all resources):

terraform show

Display the output (from outputs.tf)

terraform output

Later, we may want to destroy the infrastructure. It is straightforward to do it by executing

terraform destroy

and confirming “yes.”

To destroy all resources without the confirmation prompt:

terraform destroy -auto-approve

If you keep your “terraform” folder and the files there, you can restore your infrastructure on the cloud later. You can continue the destroy-restore cycle until you get bored!

We have finished *Terraform* except for one issue we’ll address in the Troubleshooting section. Now, we’ll see a short and delayed topic: *Boto3*.

**Boto3**

We are not going to discuss *Boto3*. In the previous articles, we mentioned the utility script for AWS a few times.

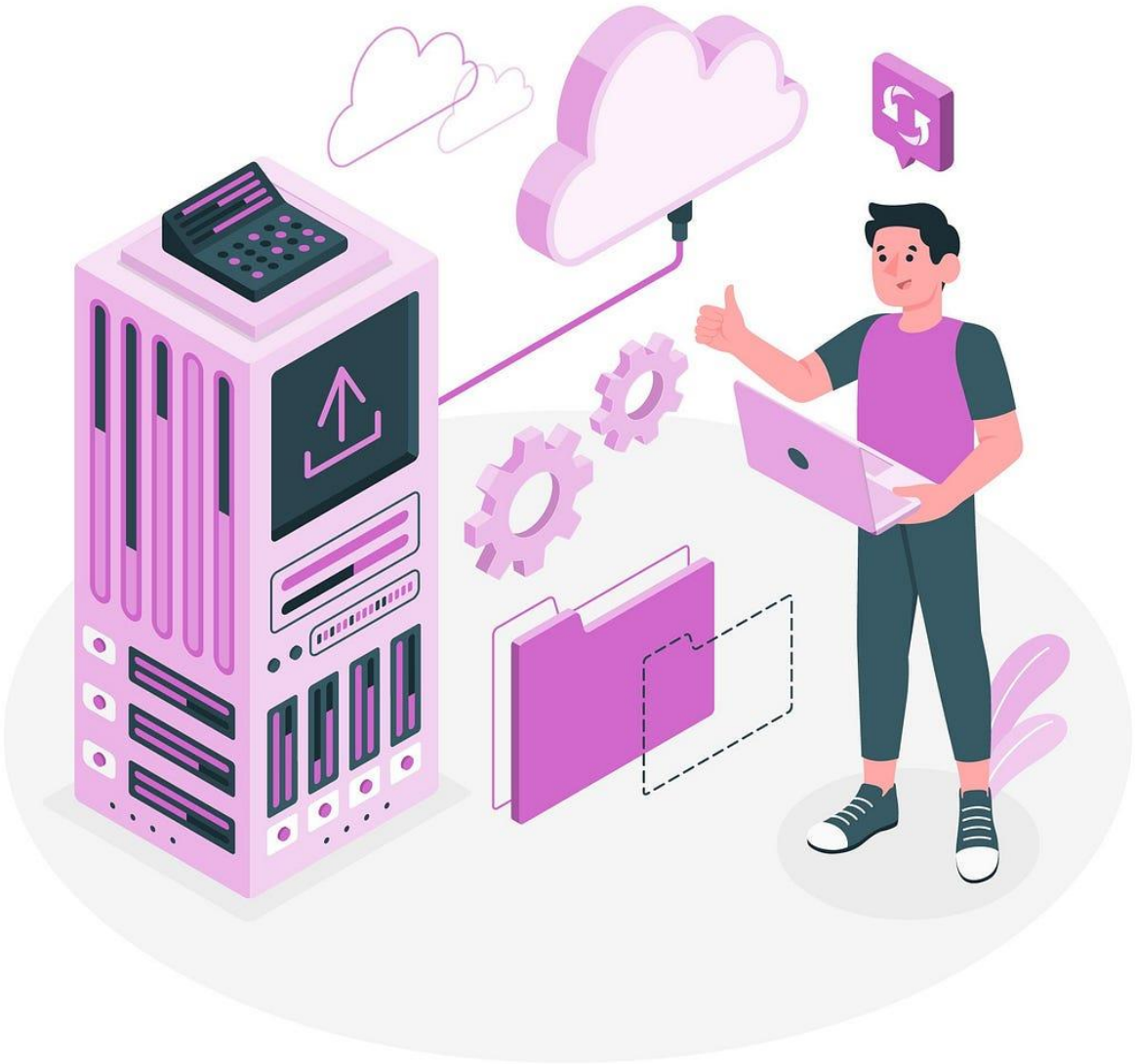


Image by storyset on Freepik

We'll see two methods here, which will help explain *Boto3*. We start by looking at how we send an email message with *Boto3*.

```
# Import libraries
from typing import IO, Any, Union

import boto3


region: str = "eu-west-1"


def send_sns_topic_message(topic_arn: str, message: str, subject: str) -> str:

    """
```

```
    Sends a notification to subscribers by email.
    """

    response = boto3.client("sns", region_name=region).publish(
        TopicArn=topic_arn,
        Message=message,
        Subject=subject,
        MessageStructure="string",
    )

    message_id = response["MessageId"]

    return message_id
```

We use a low-level client representing Amazon Simple Notification Service (SNS). SNS sends a message to an Amazon SNS topic, a text message (SMS message) directly to a phone number, or a message to a mobile platform endpoint[13]. In our project, we send a message to a topic.

“Amazon SNS delivers the message to each endpoint that is subscribed to the topic. When a messageId is returned, the message is saved and Amazon SNS immediately delivers it to subscribers.”[13]

To publish a message, we first need to create a topic. Each topic has an Amazon Resource Name (ARN). The publish method of SNS takes the topic ARN, our message text, the subject and the structure of the message as parameters. When our message text is a string, the message structure will be a string (default). If we set MessageStructure to JSON, the value of the Message parameter must be a syntactically valid JSON object[13].

Another case of using *Boto3* is when we want to get an object from the S3 bucket:

```
def get_bucket_object(bucket_name: str, object_key: str) -> tuple[IO[str], str]:

    """
    Imports or gets an object from an S3 bucket.
    """

    response = boto3.client("s3").get_object(
        Bucket=bucket_name,
        Key=object_key,
    )

    file_object = response["Body"]
    object_etag = response["ETag"]

    return file_object, object_etag
```

The get_object method of a low-level client representing Amazon Simple Storage Service (S3) retrieves the object from the S3 bucket. The object is defined by Bucket where it stays and the Key that specifies its folder and file name. The returned response provides us with the file

and an entity tag (ETag), which is an opaque identifier assigned by a web server to a specific version of a resource found at a URL[14].

We can find the other methods in the aws_utils.py file here:

**Machine-Learning/aws_utils.py at main · hsaltan/Machine-Learning**

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

github.com

For more information about *Boto3*, one may refer to their documentation which explains everything in great detail.

*Terraform* and *Boto3* are complementary tools. When starting the basic structure, *Terraform* helps handle everything with a few commands. When something needs to be done during the runtime or the code needs to behave dynamically in response to user requests, *Boto3* is the right tool.

We completed the right and middle blocks in Exhibit 1. On the left part, only *GitHub Actions* is left. In the following article, we will discuss *GitHub Action*s along with *Flask* UI, the start.sh file, Makefile , and a few other topics, and then close the series.

**Troubleshooting**

Suppose you destroyed the infrastructure, and decided to restore it later. During rebuilding, if you receive an error message related to one or more AWS resources, such as InvalidGroupId.Malformed or InvalidVpcID.NotFound upon the plan command without any valid reason, and you have successfully run the same code initially, then check the terraform.tfstate file. *Terraform* likely destroyed some resources, but not all, and the “state” file still keeps the pre-destruction state partially. So, in the “terraform” folder, execute the below commands, and delete the state files.

rm terraform.tfstate terraform.tfstate.backup

As we have seen before, *Terraform* uses the state file (terraform.tfstate) to keep track of resources and metadata information about our infrastructure.

By default, a backup of your state file is written to terraform.tfstate.backup in case the state file is lost or corrupted to simplify recovery.

Since we have already destroyed all our infrastructure, we don’t need these two files and can delete them to fix the error. When restoring, *Terraform* will already create them again.

**Part 9: Building Web UI with Flask, Deploying the Code into AWS with GitHub Actions, Connecting to the Remote Server, Executing Shell Script and Makefile, and Running the Application**

Part 8 demonstrated how to build the infrastructure on AWS with *Terraform*. Finally, we have arrived at the end of the series. In this article, we'll see how to create web UI, deploy our code into AWS, connect to the remote server, run the shell script and Makefile, create databases on *PostgreSQL*, and start and run the application from our local machine.

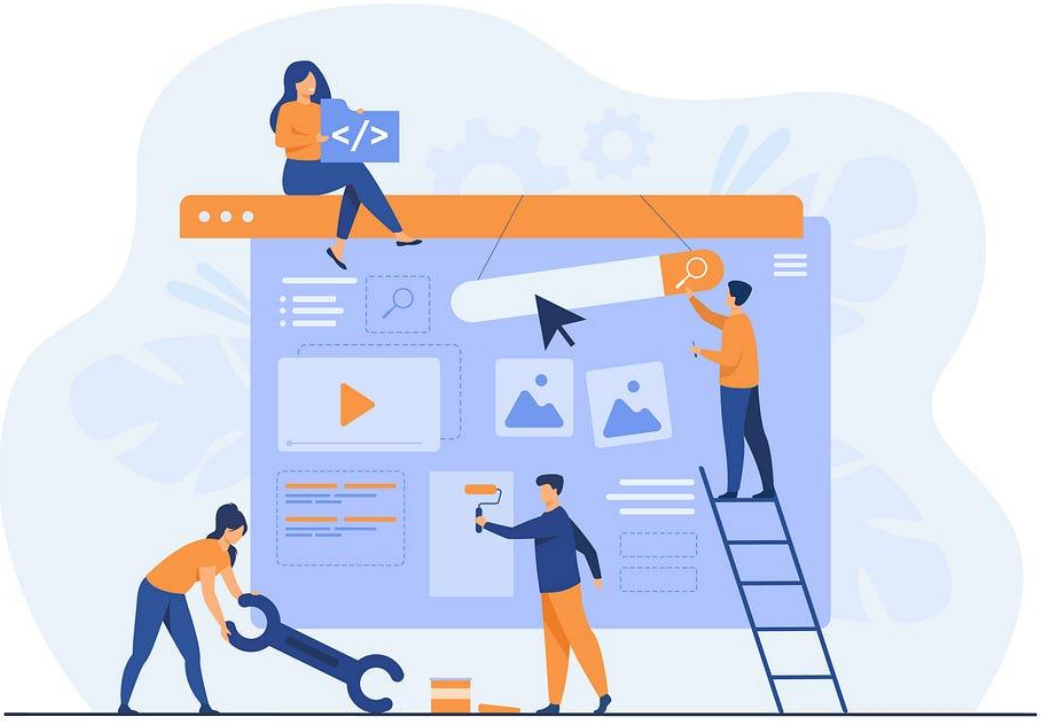


Image by pch.vector on Freepik

Our first topic is *GitHub Actions*.

**GitHub Actions**

According to the GitHub document, "GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production."[1]

*A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked in to your repository and will run when triggered by an event in your repository, or they can be triggered manually, or at a defined schedule.*

*Workflows are defined in the .github/workflows directory in a repository, and a repository can have multiple workflows, each of which can perform a different set of tasks.[1]*

*GitHub Actions* is triggered when we push our code to *GitHub*. Our workflow, behaving differently from that given in the above definition, will not test the code committed. Instead, it will upload our code and files into the EC2 instance and S3. Such a workflow might not sound like the best configuration. Generally, EC2 instances pull all data from the S3 bucket with its user data commands. Since we have done the tests during pre-commit, the proposed workflow becomes good-to-go to understand *GitHub Actions* at work.

There are steps to put *GitHub Actions* into work:

1. Create a *GitHub* repository on *GitHub* if you don't have one. In the local environment, our project folder is "Waiter-Tips-Prediction." The *GitHub* repo name must be the same as the local project directory name, which is "Waiter-Tips-Prediction."
2. Open the terminal in the project directory (in the local directory your repository is stored at) and initialize git:

```
git init
```

3. Stage and commit the project files:

```
git add .
git commit -m 'initial commit'
```

4. Add a new remote:

```
git remote add origin https://github.com/<USER>/Waiter-Tips-Prediction.git
```

5. The above command adds a remote repo but does not provide any feedback to the terminal window. To verify that the remote repo was added to our configuration, execute:

```
git remote -v
```

6. Get a personal access token from the *GitHub* Console following the instructions on this page.

7. Push the files to the remote repo. Enter the username and personal access token:

```
git push origin main
<username>
<personal access token>
```

8. Create a .github/workflows directory in your *GitHub* and local repo if they don't exist. This directory should be in the repo directory ("Waiter-Tips-Prediction"), not in any subfolders on the local machine. Creating one in the local directory, then doing the commit again will push it to the remote repo.

9. In the .github/workflows directory on the local machine, create a .yaml file, for example, github-actions-ec2-s3.yml. We'll get into the details of this file later.

10. It is not secure to explicitly declare secret information, such as passwords, keys, etc., in the code. The .yaml file we created in the previous step contains the code used to deploy the project files into AWS. In doing so, it needs particular credentials such as the *GitHub* username, EC2 ssh key, AWS access key, etc. Encrypted secrets on *GitHub* allow us to store such sensitive information in our repository or repository environments. We should create seven secrets for this project. The first one is EC2_SSH_KEY.

- EC2_SSH_KEY: When we launch projects from a repository on *GitHub* to our server, we use a “deploy key,” which is an SSH key that grants access to a single repository. “GitHub attaches the public part of the key directly to your repository instead of a personal account, and the private part of the key remains on your server.”[2]

We should follow the instructions specified in the “Set up deploy keys” section in the *GitHub* document to create a “deploy key.”

EC2_SSH_KEY will be our ssh private key file which we will use to log in to the instance. We will copy the key inside the wtpkey file and paste it into *GitHub* “Key” section. Pasting should occur in the section after beginning with the digital signature system, such as ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384, ecdsa-sha2-nistp521, ssh-ed25519, sk-ecdsa-sha2-nistp256@openssh.com, or sk-ssh-ed25519@openssh.com. If you recall, we created our key in Part 8 as the ed25519 key. So, we’ll begin with ssh-ed25519.

Our private key, wtpkey , is located in the .ssh folder on our local machine. The block that will be copied and pasted in the *GitHub* “Key” section is the segment between BEGIN and END declarations.

-----BEGIN OPENSSH PRIVATE KEY-----
1hYy5sb2NhbAXktdjEAAAAABG5vbmUAAAAEMSah35lTJHfd0V533SYwAAAKgWG7YCFhu2S
QyNTUxOQAAACCBsqZ00M1ZmgLFCMMVkAAAEA2Fbu9Hiyw1kTQAyFlEW2XbGWG7YCFhu200
AgAAAAtzc2gtZWQyNTUxOQAAACCBsqZ00M1ZmgLFCMMVk6HLIZMSah35lTJHfd0V533SYw
AAAtzc2gtZWQyNTU93RXnfdJjAAAATi3q6hKeVfUBlWUoGypnTQzVmaAsUIwxWTocshSGT
kxJqHfmVMkd93RXnfdJjAAAAIWhhc2Fuc2VyZGFyYWx0YW5ASGFzYW4taU1hYy5sb2NhbA
ECAwQ=
-----END OPENSSH PRIVATE KEY-----

Lastly, we should leave “Allow write access” as checked to enable the key to be used to push to the repository. By doing so, we don’t need to give GitHub the username and password every time. Deploy keys always have “pull” access.

Once we enter all this information in the Deploy Key, we’ll see the following screen:

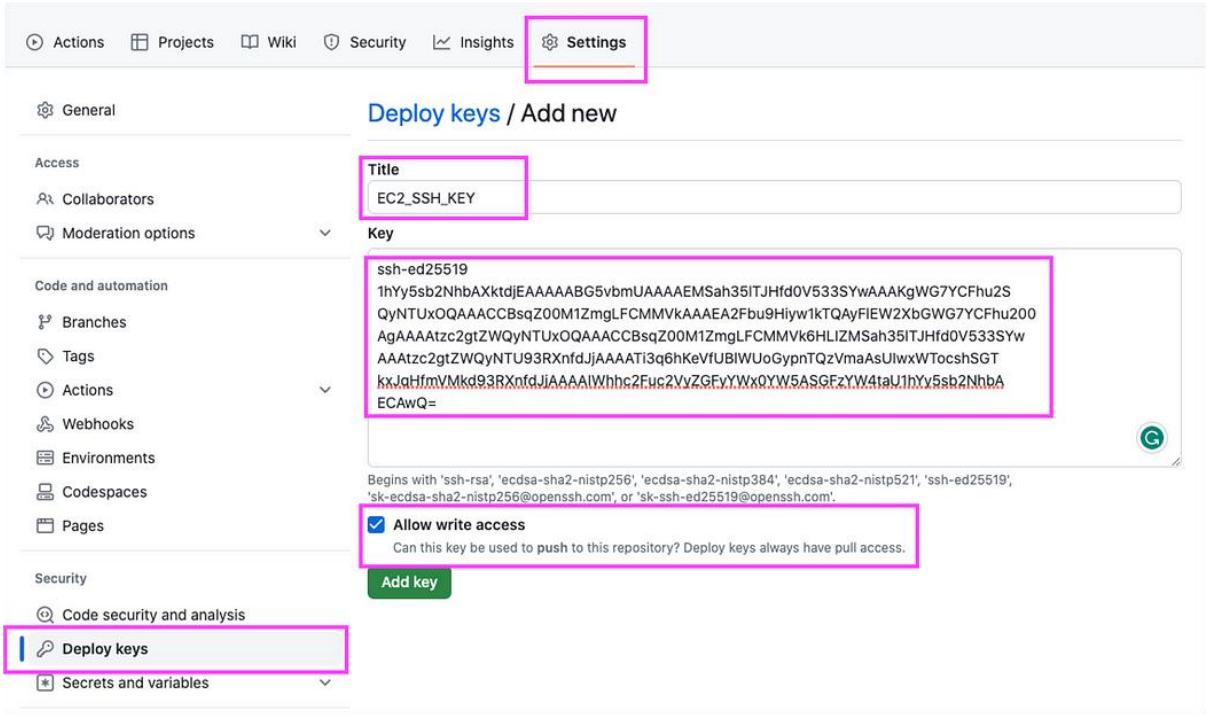



Exhibit-1: Setting up the Deploy Key (Image by Author)

We will create each of the remaining six secrets by following the steps under the "Creating encrypted secrets for a repository" section on the *GitHub* page. Secrets are encrypted before they reach *GitHub* and remain encrypted until we use them in a workflow[3].

- HOST_DNS: Public DNS record of the instance. It looks like ec2-xx-xxx-xxx-xxx.us-west-2.compute.amazonaws.com. We get this DNS only after *Terraform* deploys all resources. We have already done it.
- USERNAME: It is the username of the EC2 instance, which is ubuntu in our case.
- TARGET_DIR: This variable specifies where we want to deploy our code and files on the EC2 instance. It is /home/ubuntu/app.
- AWS_ACCESS_KEY_ID: This is our AWS access key ID.
- AWS_SECRET_ACCESS_KEY: It is our secret access key. We have obtained it and the access key ID by acquiring the programmatic access during the AWS IAM user creation.
- AWS_BUCKET: It is the S3 bucket and key, which house our data files, which is s3://s3b-tip-predictor/data/.

After setting up the secrets according to the instructions in the *GitHub* pages given, we may now look at the contents of the github-actions-ec2-s3.yml file:

```
name: Push-to-EC2 and Upload to S3

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

```yaml
jobs:
  deploy:
    name: Deploy to EC2
    runs-on: ubuntu-latest

    steps:
      - name: Main Branch
        uses: actions/checkout@main

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: "eu-west-1"

      - name: Deploy Code to EC2
        uses: appleboy/scp-action@master
        with:
          source: "./Waiter-Tips-Prediction"
          key: ${{ secrets.EC2_SSH_KEY }}
          host: ${{ secrets.HOST_DNS }}
          username: ${{ secrets.USERNAME }}
          port: 22
          target: ${{ secrets.TARGET_DIR }}

  # Upload file to S3
  upload:
    name: Upload to S3
    runs-on: ubuntu-latest

    steps:
      - name: Main Branch
        uses: actions/checkout@main

      - name: Upload File to Bucket
        uses: zdurham/s3-upload-github-action@master
        with:
          args: --acl public-read
        env:
          AWS_REGION: "eu-west-1"
          S3_BUCKET: ${{ secrets.AWS_BUCKET }}
          S3_KEY: "data/tips.csv"
          FILE: "Waiter-Tips-Prediction/data/tips.csv"
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

*GitHub Actions*, by this file, enables us to upload the project code and files to the EC2 instance and the raw data to the S3 bucket. Therefore, whenever anything changes in the files in our project directory on the local machine, it is enough for us to commit and push the files to the remote repo. *GitHub Actions* will upload the new version files and data to their target destinations.

All pull and push requests occur on the main branch. *GitHub* provides a Linux virtual machine to run our workflow. More specifically, the workflow runs on the latest version of Ubuntu. *GitHub* calls these machines "runners." Therefore, a runner is a server that runs our workflows when they're triggered. Each runner can run a single job at a time[1].

We have two jobs: 1. Deploying the code to EC2, 2. Uploading the file to the S3 bucket.

A job consists of steps that get executed on the same runner. Each step is a shell script or an action. Steps are sequentially executed and dependent on each other. Data can be shared among the steps since they are executed on the same runner. Jobs, by default, have no dependencies but can be configured to have dependencies and run in parallel. When a job depends on another one, it will wait for the other job to complete before it runs[1].

For both jobs, we need to define our AWS credentials (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and AWS_REGION), also the target endpoint details, such as the ssh key, hostname, username, port, and the target directory for the EC2 instance, and the bucket name and key for S3. We import these variables we created earlier from the secret store of *GitHub*.

The source of the files for the EC2 instance is the ./Waiter-Tips-Prediction directory and is defined with the source parameter. The source of the data for the S3 bucket is Waiter-Tips-Prediction/data/tips.csv and specified with the FILE parameter.

In each step of the jobs, we run an action, which is specified with the uses parameter. "An *action* is a custom application for the GitHub Actions platform that performs a complex but frequently repeated task. Use an action to help reduce the amount of repetitive code that you write in your workflow files."[1] For example, aws-actions/configure-aws-credentials@v1 we used helps us configure our AWS credentials and region environment variables for use in other *GitHub Actions*.

We can write our actions or find them written by others in the GitHub Marketplace.

We have composed the github-actions-ec2-s3.yml file, so we can stage and commit it:

```
git add .
git commit -m 'commit actions yaml file'
```

Finally, we push the .github/workflows directory and the YAML file to the remote repo, entering the username and personal access token:

```
git push origin main
<username>
<personal access token>
```

Later when we commit and push a change in the local directory, it will trigger *GitHub Actions* and upload the updated files to EC2 and S3.

After *GitHub Actions* is installed, we may need to update the *GitHub* Secrets if any change is made on AWS resources, e.g., stopping or terminating the instance. The new EC2 instance will have a different DNS name.

The .github/workflows folder resides in the top directory. Therefore, when we commit any change in the project directory, it will trigger the *GitHub Actions*. If you create another project in the ./Waiter-Tips-Prediction directory on the local machine, *GitHub Actions* will try to upload the files of this new project to the EC2 instance and S3 bucket. However, the latest project may not have anything to do with the said AWS resources. To prevent this, we may either create another remote repo (also another local repo with the same name) for the new project or temporarily suspend the *GitHub Actions*. While the latter doesn't seem like a practical solution, if we insist on it, we can do it by performing the steps described in the documentation.

We uploaded all files and data to the AWS resources on our infrastructure, and now, we can ssh into our remote server by following the steps below:

1. Open the terminal, change the current directory to the ~/.ssh directory, and open the config file.

```
cd ~/.ssh
nano config
```

2. Ensure the HostName (3.249.105.159) matches the current public IP address of our server. After every stop-and-start operation, our instance will have a new public IP address.

```
Host wtp
  HostName 3.249.105.159
  User ubuntu
  IdentityFile ~/.ssh/wtpkey
  ServerAliveInterval 60
  StrictHostKeyChecking no
```

3. Append ServerAliveInterval 60 and StrictHostKeyChecking no to the config file. They will prevent the SSH connection from being broken often. Save and exit the file.

4. SSH into the server.

```
ssh wtp
```

We want to fulfill the last step not on the local terminal window but on VSC. For this, we need to connect to our remote server from VSC.

**Connecting to the Remote Server (EC2)**

We should execute scripts, install packages, and do some work on the remote server. In addition, we may want to perform some database operations. When we launch our RDBMS server, we can see the databases on *PGAdmin* or the browser of our local machine. For the latter, we need to do port forwarding on VSC.

So far, we have used several ports for different services:

- Grafana: 3000
- Flask application: 3500 and 3600

- Mflow: 5000
- Evidently: 5500
- Airflow web server: 8080
- Airflow scheduler: 8793
- Prometheus: 9090 and 9091

To perform all jobs mentioned above, we need to connect to the remote server from VSC. Before doing that, ensure you have fulfilled the following steps:

1. Create an AWS account.
2. Choose “Python Select Interpreter” on VSC, then “Python 3.9.9.”
3. Install AWS Kit on VSC.
4. Create AWS Profile on VSC. Enter aws_access_key_id and aws_secret_access_key.
5. Set the region by editing the config file, which is located in ~/.aws/config.

We have already done this part during the *Terraform* deployment. Now, we are ready to start the connection procedure by performing the following steps:

1. Install the Remote-SSH extension on VSC.

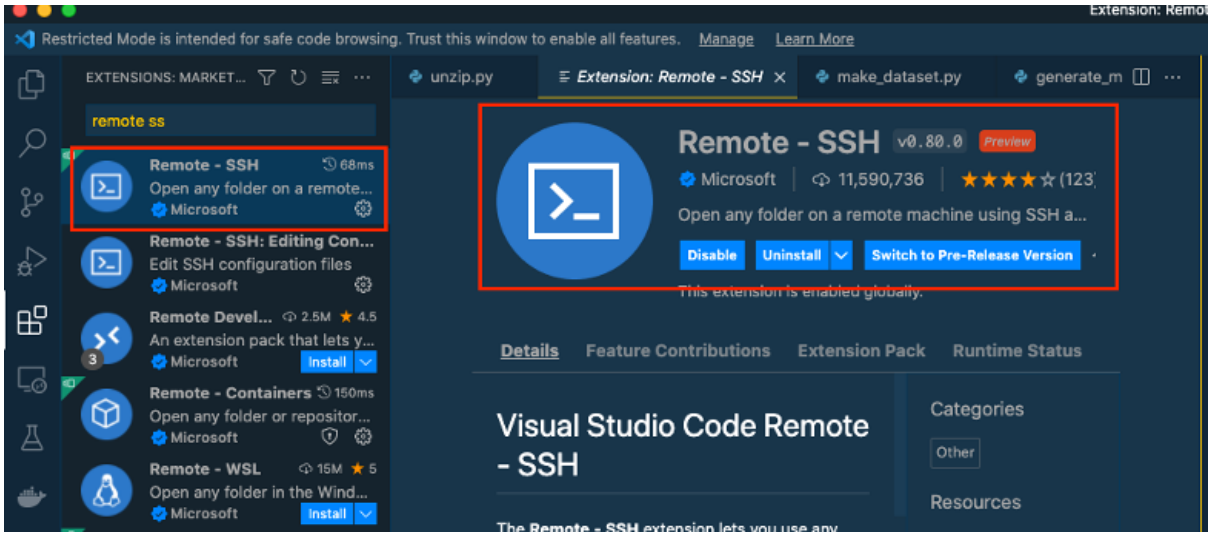



Exhibit-2: Installing Remote-SSH Extension on VSC (Image by Author)

2. Click the “Remote Explorer” on the left, and click “+” to add a new SSH target. Type “ssh wtp” to connect. Choose the config file in your local directory.

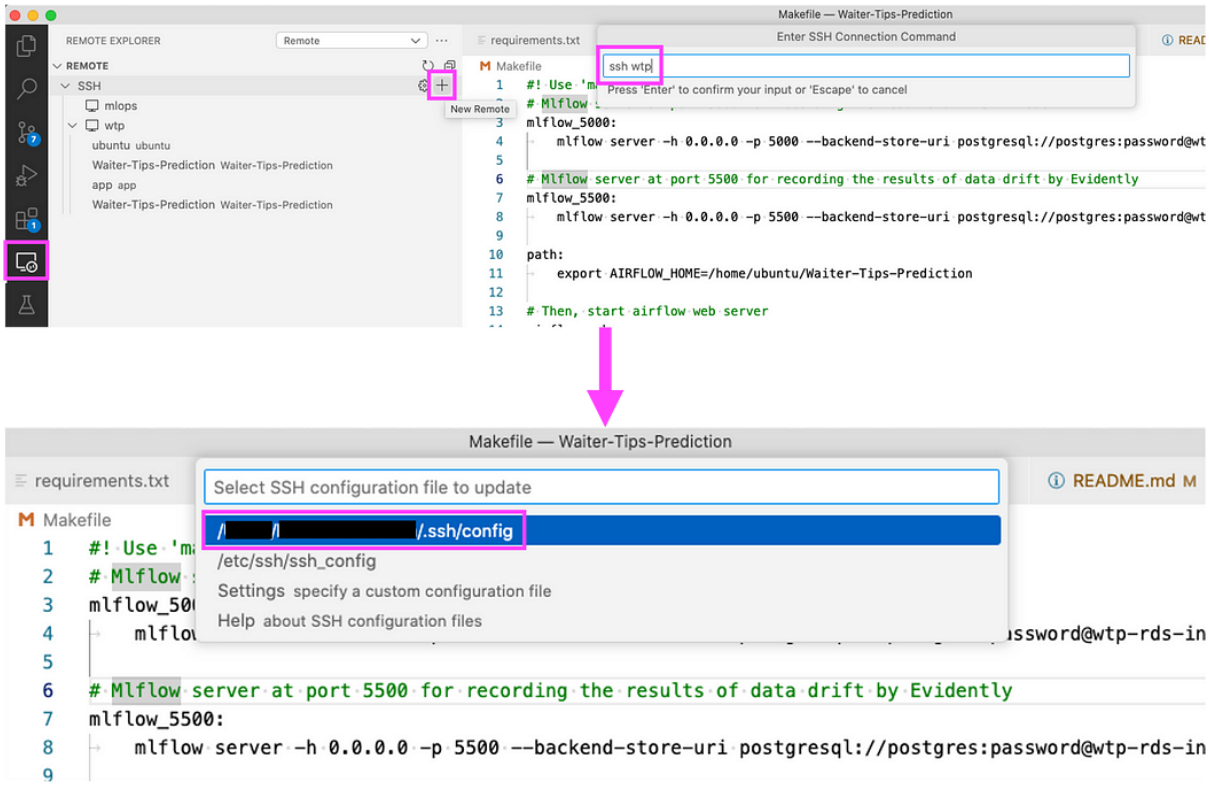



Exhibit-3: Remote-SSH Connection on VSC (Image by Author)

The information in the config file must match the connection details of your active EC2 instance.

3. On the “Remote Explorer,” click the folder icon next to our hostname to connect to the remote server. We want to open the server in a new window.

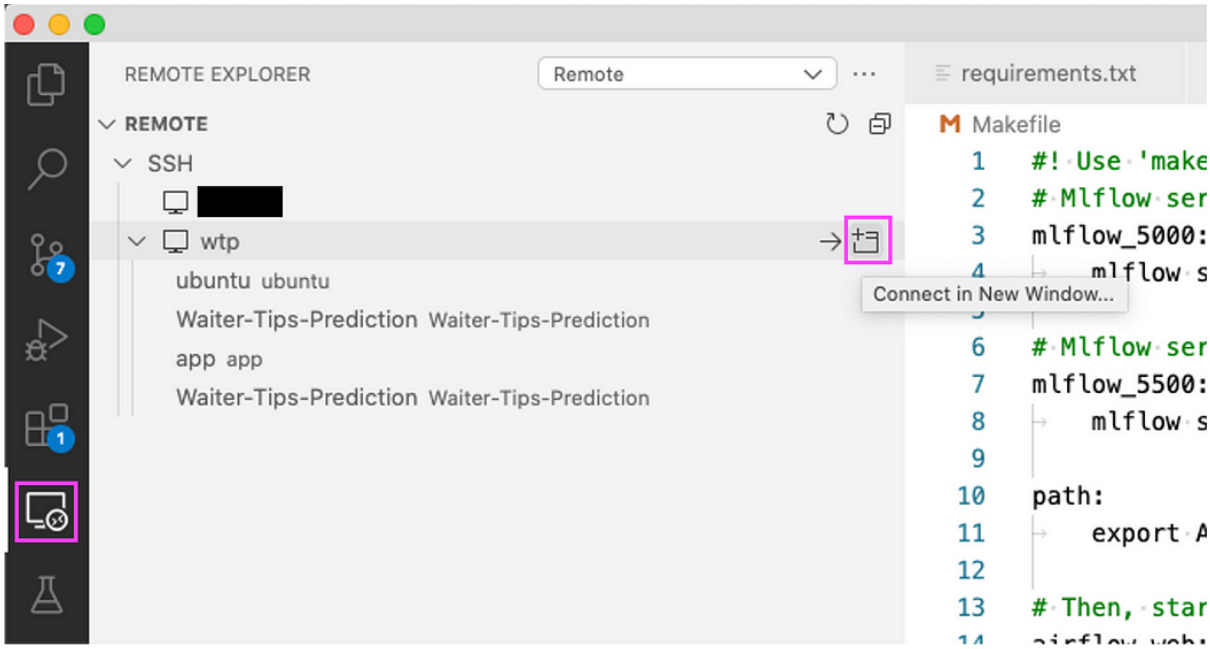



Exhibit-4: Opening the Remote Server on VSC (Image by Author)

4. Click “Open…”, choose “/home/ubuntu/” and click “OK.”

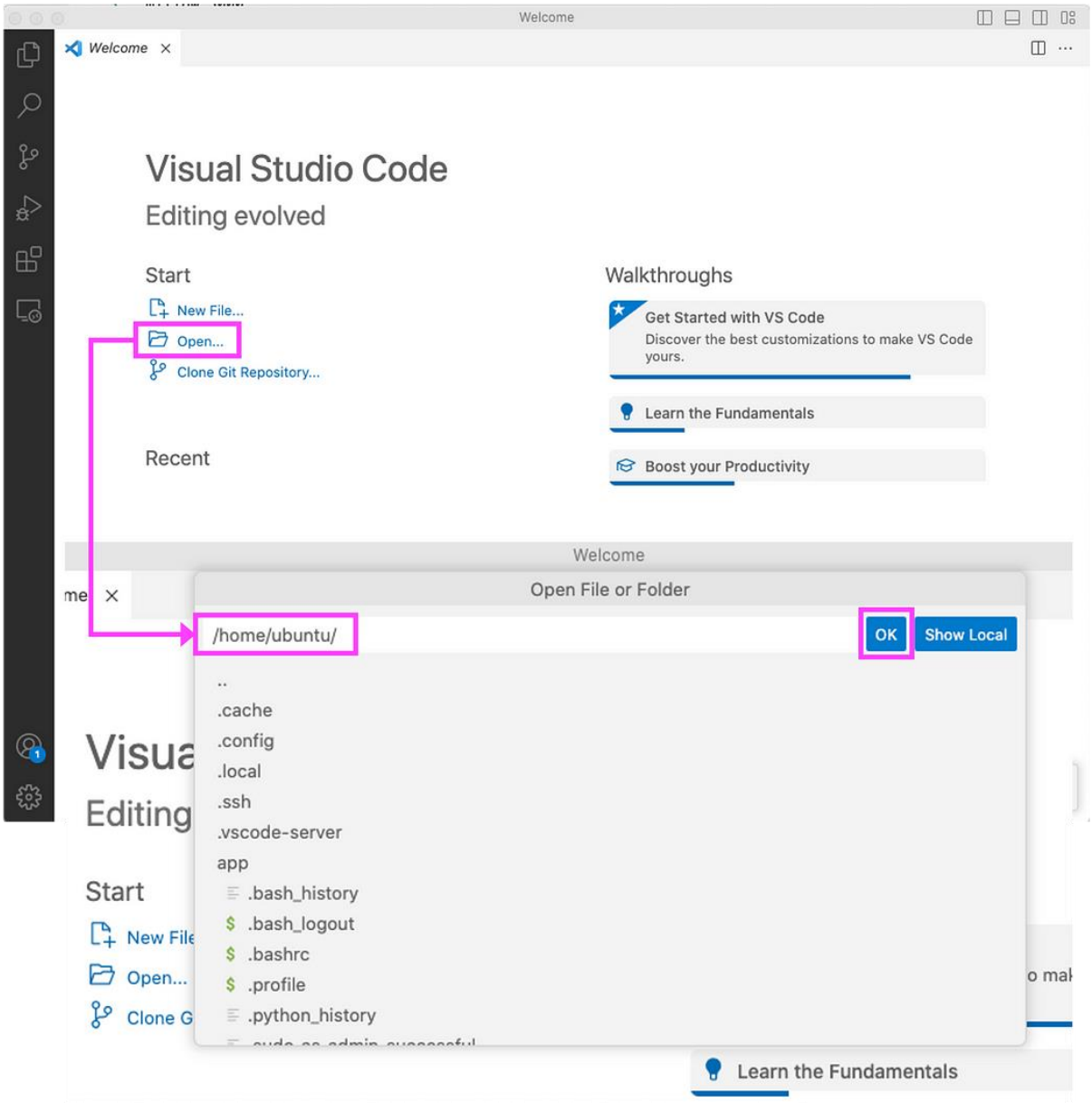



Exhibit-5: Opening the Directory on the Remote Server (Image by Author)

5. Choose “Yes, I trust the authors.”

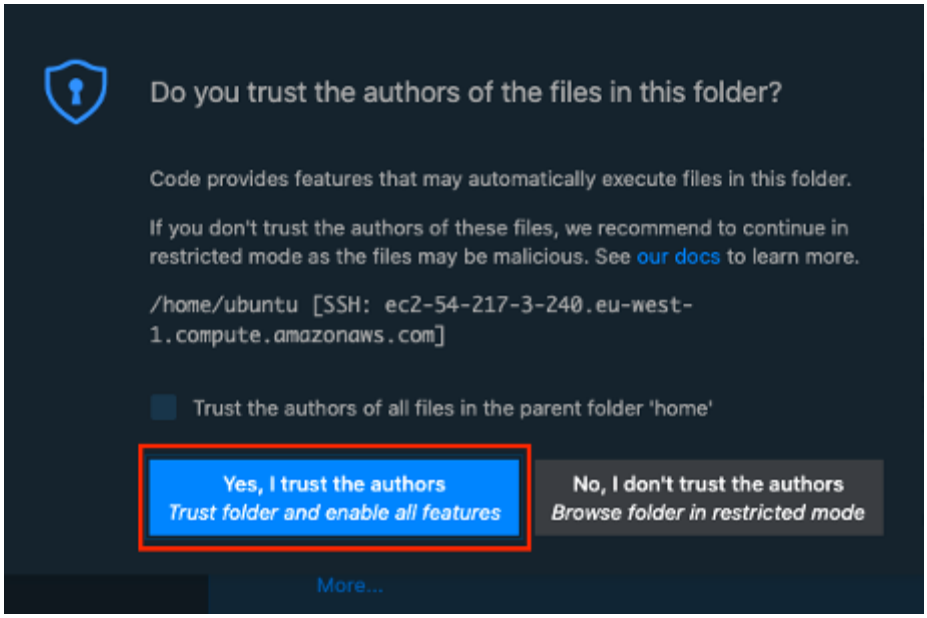



Exhibit-6: Trusting Folder and Enabling All Features (Image by Author)

This step will open the terminal in the /home/ubuntu directory on the EC2 instance.

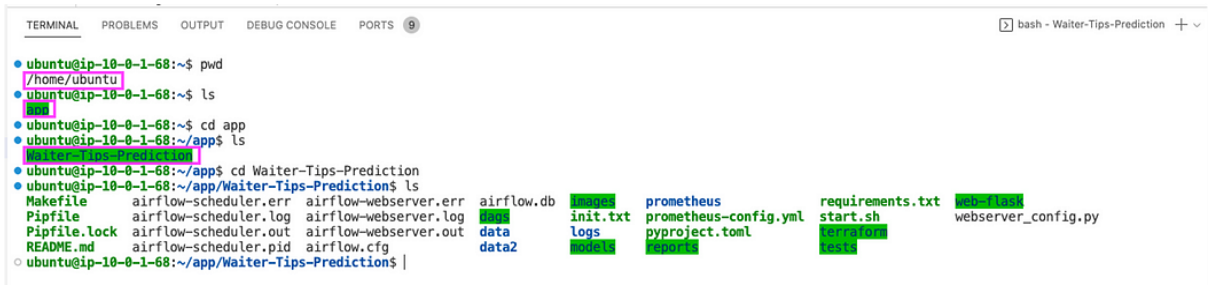



Exhibit-7: The Terminal Opening in the Parent Directory of the Application (Image by Author)

We have connected to the remote server and will now do port forwarding.

6. Choose the “Ports” tab and click “Add Port” to add all ports we need to open pages in the browser.

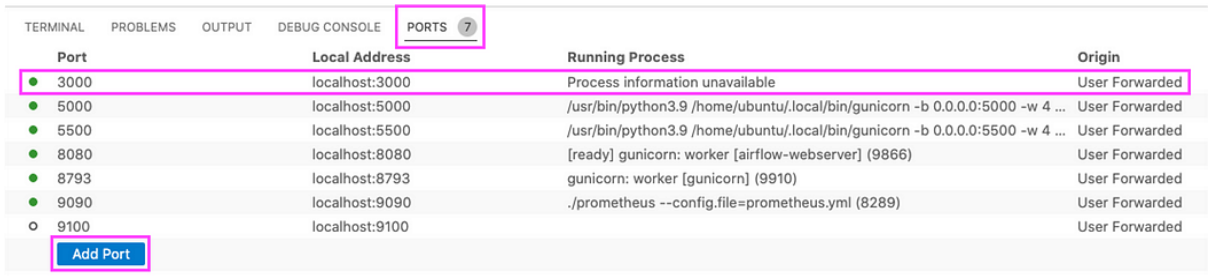



Exhibit-8: Port Forwarding (Image by Author)

Now, we can open all the pages in the browser of our local machine. For example, we type http://0.0.0.0:5000 in the browser to open the web UI of *MLFlow*. Similarly, http://0.0.0.0:8080 will open the web UI of *Airflow*.

Next, we will complete the remaining installation and start the services on the terminal of our remote server.

**Executing Shell Script and Makefile**

In this section, we'll see two files to execute: start.sh and Makefile. There are a couple of preliminary adjustments to make.

Change the root user password on our Ubuntu machine:

sudo passwd ubuntu

At the prompt, enter the new password (ubuntu), and confirm it. This operation may not be necessary for any other Linux machine. Ubuntu may ask for the password before installing Homebrew.

*start.sh*

This file contains the instructions to complete the remaining installation. The entire content of the start.sh file can be found here:

**Machine-Learning/start.sh at main · hsaltan/Machine-Learning**

This is about machine learning projects. Contribute to hsaltan/Machine-Learning development by creating an account on...

github.com

Let us look at the commands in the start.sh file one by one.

1. Change the current directory to ./app/Waiter-Tips-Prediction.

```
#!/bin/bash
cd ./app/Waiter-Tips-Prediction
```

2. Install Homebrew, and run the subsequent commands as the installation instructions will describe on the screen.

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
echo '# Set PATH, MANPATH, etc., for Homebrew.' >> /home/ubuntu/.profile
echo 'eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv)"' >> /home/ubuntu/.profile
eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv)"
```

3. Install the build-essential package, which is a link to several other packages that will be installed as dependencies.

sudo apt-get install build-essential

4. Install gcc, which is a set of compilers and development tools available for Linux, Windows, various BSDs, and a wide assortment of other operating systems.

brew install gcc

5. Install and start *Prometheus*.

```
brew install prometheus
brew services restart prometheus
```

6. Copy the prometheus-config.yml file to the /home/linuxbrew/.linuxbrew/etc directory as prometheus.yml , restart the *Prometheus* service, and get status info.

```
cp -f /home/ubuntu/app/Waiter-Tips-Prediction/prometheus-config.yml
/home/linuxbrew/.linuxbrew/etc/prometheus.yml
brew services restart prometheus
brew services info prometheus
```

7. Wait for five seconds and get the status info for the *PostgreSQL* service, and wait for another five seconds to get the status info for the *Grafana* service.

```
sleep 5
sudo systemctl status postgresql.service --no-pager
sleep 5
sudo systemctl status grafana-server --no-pager
```

8. After waiting for five seconds, install and uninstall the following packages to maintain compatibility.

```
sleep 5
pip install chardet==4.0.0
pip install requests -U
pip install -U click
pip uninstall Flask-WTF -y
pip uninstall  WTForms -y
pip install Flask-WTF==0.15.1
pip install  WTForms==2.3.3
```

We addressed this compatibility issue in the Troubleshooting section of Part 1.

9. Move the *Airflow* home directory to the project directory.

```
echo 'export AIRFLOW_HOME=/home/ubuntu/app/Waiter-Tips-Prediction'
```

10. Log in to our *PostgreSQL* database.

```
sudo -i -u postgres
```

When we install *PostgreSQL*, a default admin user, “postgres” is created. We must use it to log in to our *PostgreSQL* database for the first time.

11. Connect to the RDS database. We can get the RDS connection details after *Terraform* builds it.

```
psql \
  --host=wtp-rds-instance.cmpdlb9srhwd.eu-west-1.rds.amazonaws.com  \
  --port=5432 \
  --username=postgres \
  --password \
  --dbname=mlflow_db <<EOF
CREATE DATABASE mlflow;
```

```
CREATE DATABASE airflow;
CREATE DATABASE evidently;
\l
EOF
```

At the prompt, enter the password (password). CREATE commands will create three more databases. They will store metrics and information of *MLFlow*, *Airflow,* and *Evidently*. The \l command lists the existing databases.

We cannot ssh into RDS. A more suitable way to observe the server and databases is using *PGAdmin*. After downloading *PGAdmin*, we open the application and right-click the "Server" item.

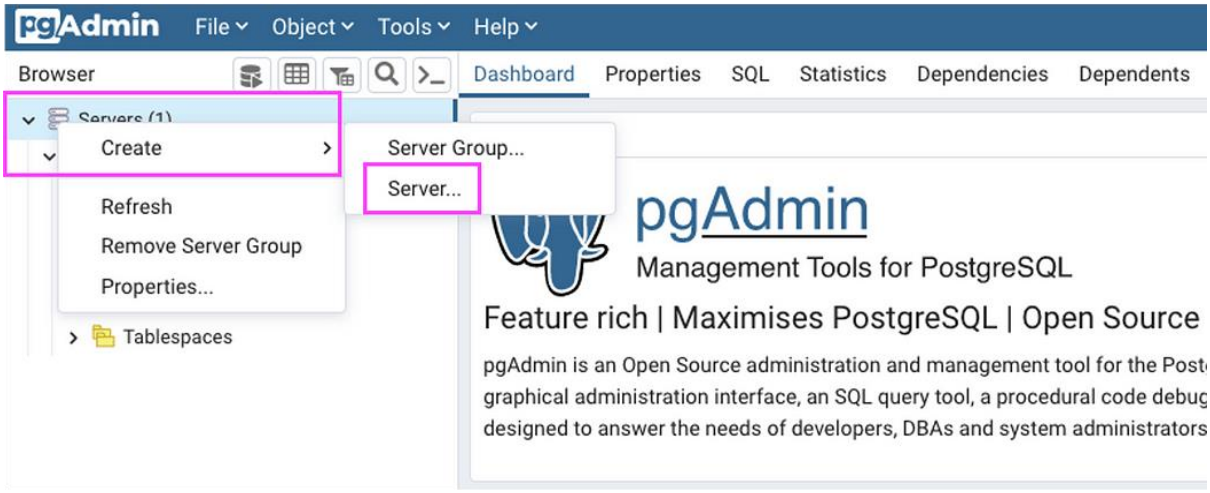



Exhibit-9: Creating Server for RDS on PGAdmin (Image by Author)

We choose "Server" and enter the connection details of our RDS in the related cells in the "General" and "Connection" tabs, and save them.

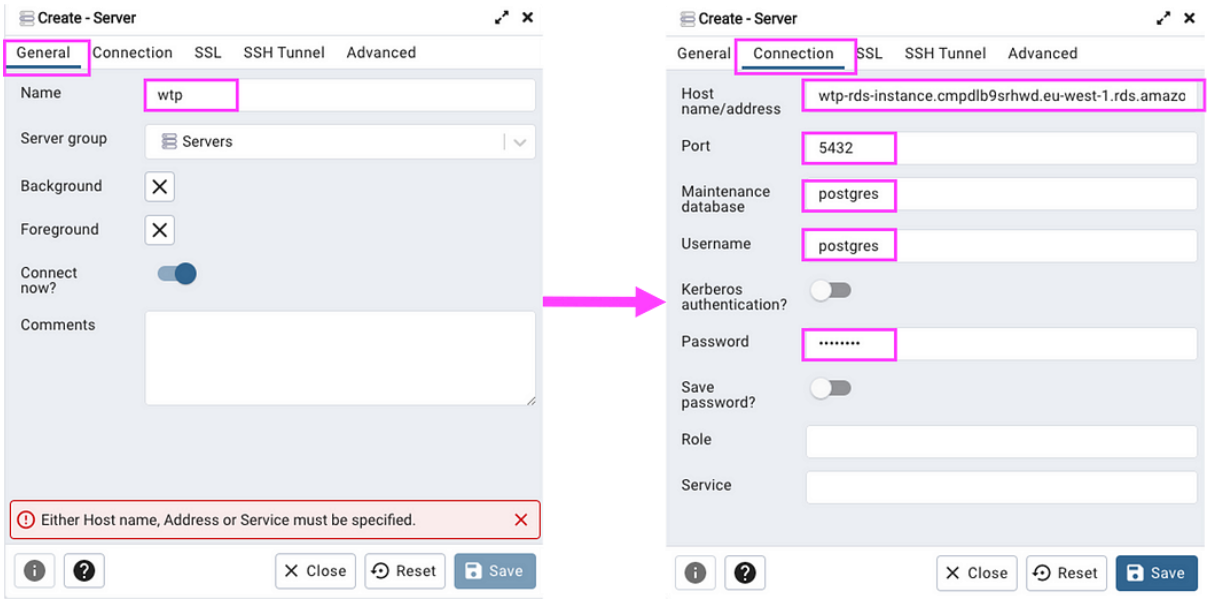



Exhibit-10: Entering the Connection Information of RDS in PGAdmin (Image by Author)

After creating the server on *PGAdmin* to connect to the RDS, we right-click the active server icon on the left panel and choose “Database…”, then give a name to the new database and click “Save” to create the new one in the server.

Using *PGAdmin* UI makes it easier to create more databases after the installation unless we want to do it by a shell script file such as start.sh.

We can execute all the above steps in order by running start.sh file. We made this file executable during the launch of EC2 through the user data.

./start.sh

Execution of start.sh will complete the remaining part of the installation and deployment.

*Makefile*

Another file we use is Makefile. Make is a utility that is designed to perform the execution of a special file that contains a set of shell commands and is named Makefile. We generally use Makefile to run or update a task when certain files are updated. In this application, Makefile will help start or terminate our *MLFlow, Evidently,* and *Airflow* servers. Before starting, ensure that make is installed in your system.

Each set of commands in the Makefile is executed by typing make followed by the name given for the set that will be run. Our Makefile is:

```
# Mlflow server at port 5000 for recording the results of the ML model
mlflow_5000:
 mlflow server -h 0.0.0.0 -p 5000 --backend-store-uri postgresql://postgres:password@wtp-rds-instance.cmpdlb9srhwd.eu-west-1.rds.amazonaws.com:5432/mlflow_db --default-artifact-root s3://s3b-tip-predictor/mlflow/

# Mlflow server at port 5500 for recording the results of data drift by Evidently
mlflow_5500:
 mlflow server -h 0.0.0.0 -p 5500 --backend-store-uri postgresql://postgres:password@wtp-rds-instance.cmpdlb9srhwd.eu-west-1.rds.amazonaws.com:5432/evidently --default-artifact-root s3://s3b-tip-predictor/evidently/

path:
 export AIRFLOW_HOME=/home/ubuntu/Waiter-Tips-Prediction

# Then, start airflow web server
airflow_web:
 airflow db init
 airflow webserver -p 8080 -D
 lsof -i tcp:8080

# If starting airflow server fails, run this
airflow_web_reset:
 rm airflow-webserver.err  airflow-webserver-monitor.pid

# Start airflow scheduler
```

```
airflow_scheduler:
 airflow scheduler -D
 lsof -i tcp:8793

# If starting airflow scheduler fails, run this
airflow_scheduler_reset:
 rm airflow-scheduler.err  airflow-scheduler.pid
```

All the commands above should look familiar from the previous articles. They represent repetitive tasks. By using Makefile, we shortcut the execution of multiple shell commands. For example, to start the *Airflow* web server, we can simply type:

```
make airflow_web
```

This statement will sequentially execute three commands listed under airflow_web in the file. A quick reminder: we should create a user before starting the *Airflow* server and scheduler. And we saw how to create a user on *Airflow* in Part 4.

When we execute the "launch" commands in Makefile, our servers will be ready to run the application, and we can see them on the browser by typing the localhost and the correct port.

The next and final topic in our project is the *Flask* application, and we'll see it in the following section.

**Web UI with Flask**

We will generate two HTML (base.html and index.html) and one *Python* file (app.py) for this simple application. In the project directory (/home/ubuntu/app/Waiter-Tips-Prediction), we first create a folder named "web-flask," and navigate into it:

```
mkdir web-flask
cd web-flask
```

Within the "web-flask" folder, we create another folder "templates":

```
mkdir templates
```

The base.html and index.html files will be located in the "templates" folder, and the app.py will reside in the "web-flask" folder. If you recall from Part 2, we also placed *Evidently* monitoring web files in the "templates" folder, and the "web-flask" directory housed the related *Python* script (evidently.py).

We use base.html file to reduce repetitive markup in other web pages. In other words, the following pages inherit particular elements from the base template and fill in the blocks in the base file with the data unique to these pages. Our base.html will look like this:

```
<!doctype html>
<html>
<head>
   <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
```

```
  <title>{% block title %}{% endblock %}</title>
</head>
<body>
  {% block content %}
  {% endblock %}
  <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
integrity="sha384-
UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous"></script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
</body>
</html>
```

Our child templates fill in two blocks defined with {% block %} tags in the base file. They are the title and the body content. The script tag content is excerpted from *Bootstrap*[4]. We have one child template, index.html:

```
{% extends "base.html" %}
{% block title %}Entry Page{% endblock %}

{% block content %}
<body style="background-color:powderblue;">
<br><br>
<h3 style="text-align: center;">Waiter Tip Prediction</h3>
<br><br>
<form action="#" method="post" align="center">
  <text style="font-size:15px; font-style: italic;">Total bill in dollars including tax:</text>
 <p><input type="text" name="nm" placeholder="e.g.12.45" required="required" style="width:
260px; height:30px; font-size:14px;"/></p>
  <text style="font-size:15px; font-style: italic;">Whether the person smoked or not:</text>
  <p><select name="smoker" method="GET" required="required" style="width: 260px;
height:30px; font-size:14px;" action="/">
    {% for each in smokers %}
      <option value="{{each}}" SELECTED>{{each}}</option>"
    {% endfor %}
 </select></p>
 <text style="font-size:15px; font-style: italic;">Gender of the person paying the bill:</text>
 <p><select name="gender" method="GET" required="required" style="width: 260px;
height:30px; font-size:14px;" action="/">
  {% for each in genders %}
    <option value="{{each}}" SELECTED>{{each}}</option>"
  {% endfor %}
  </select></p>
  <text style="font-size:15px; font-style: italic;">Day of the week:</text>
```

```
  <p><select name="week_day" method="GET" required="required" style="width: 260px;
height:30px; font-size:14px;" action="/">
    {% for each in week_days %}
      <option value="{{each}}" SELECTED>{{each}}</option>"
    {% endfor %}
  </select></p>
  <text style="font-size:15px; font-style: italic;">Lunch or dinner:</text>
  <p><select name="day_time" method="GET" required="required" style="width: 260px;
height:30px; font-size:14px;" action="/">
    {% for each in day_times %}
      <option value="{{each}}" SELECTED>{{each}}</option>"
    {% endfor %}
  </select></p>
  <text style="font-size:15px; font-style: italic;">Number of people attending:</text>
 <p><input type="text" name="rm" placeholder="e.g.5" required="required" style="width:
260px; height:30px; font-size:14px;"/></p>
 <br>
 <p><input type="submit" value="Submit" style="width: 100px; height:30px; font-
size:15px;"/></p>
</form>
<br>
<h5 style="text-align: center; color:blue;">Predicted tip amount: US$ {{ result }}</h5>
<br>
<p style="text-align:center; font-size:14px; color:blue;">
<text>Total bill (US$): {{total_bill}}</text><br>
<text>Number of people: {{size}}</text><br>
<text>Smoker or non-smoker: {{smoker_value}}</text><br>
<text>Gender: {{gender_value}}</text><br>
<text>Week day: {{week_day_value}}</text><br>
<text>Time: {{day_time_value}}</text>
</p>
</body>
{% endblock %}
```

Our index.html shows the overriding content of the body and the title blocks. Here, we define text and dropdown boxes with placeholders and insert a title and multiple-line text. Going into explaining details in the above code is beyond this article. However, we may see what it will display on the web browser. The below image shows the main page with values selected but not yet submitted:

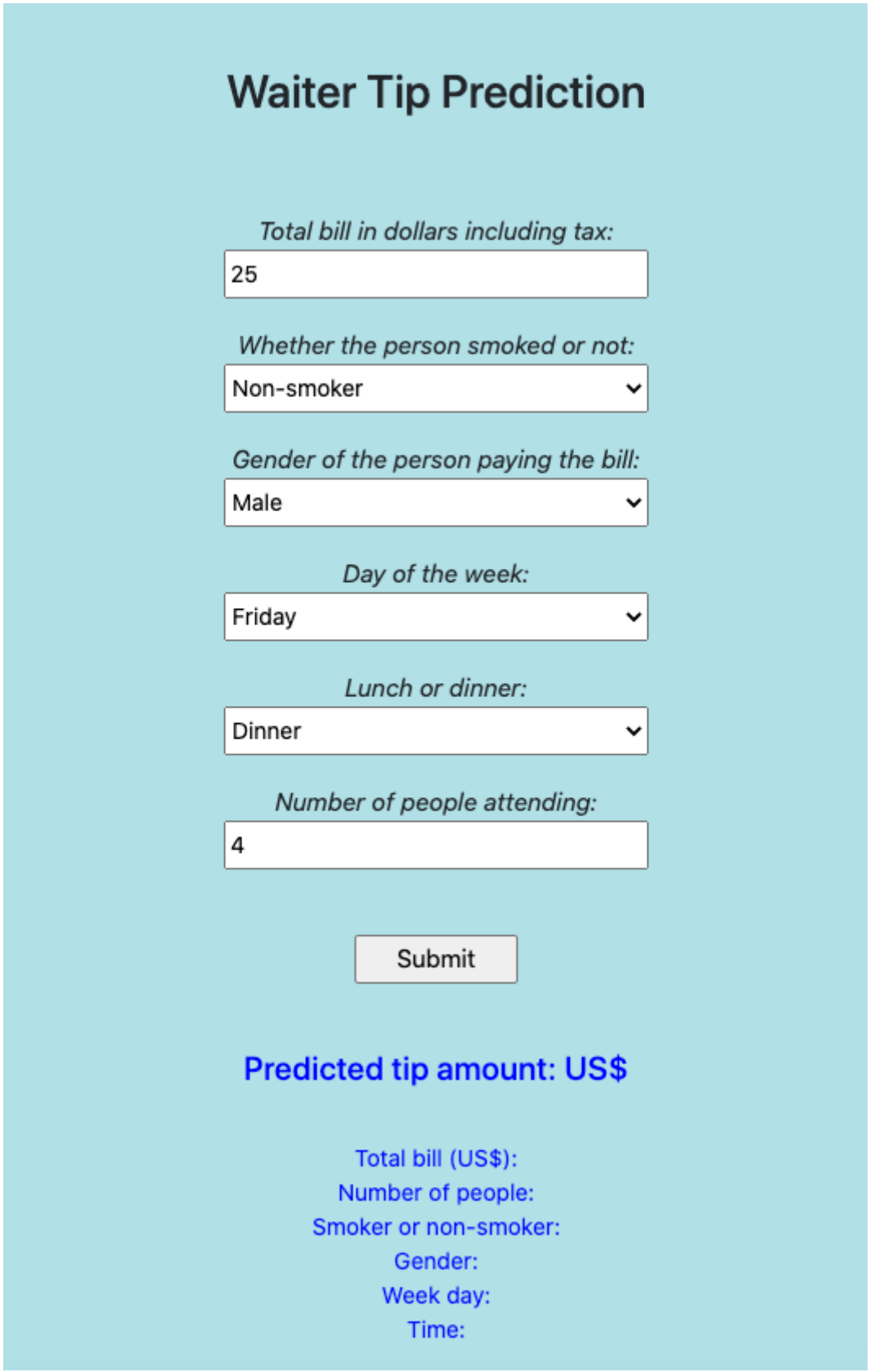



Exhibit-11: Web UI (Image by Author)

When we click the “Submit” button, it runs the app.py, which is:

```
# Import libraries
import sys

sys.path.insert(1, "/home/ubuntu/app/Waiter-Tips-Prediction/dags/app")
from flask import Flask, render_template, request
from predict import predict

app = Flask(__name__)


@app.route("/")
def my_form():

    """
    Displays the web page, text and dropdown boxes
    """

    smokers = ["Non-smoker", "Smoker"]
    genders = ["Female", "Male"]
    week_days = ["Thursday", "Friday", "Saturday", "Sunday"]
    day_times = ["Lunch", "Dinner"]

    return render_template(
        "index.html",
        smokers=smokers,
        genders=genders,
        week_days=week_days,
        day_times=day_times,
    )


@app.route("/", methods=["POST", "GET"])
def enter():

    """
    Receives user input on the web page and computes the prediction.
    """

    if request.method == "POST":
        total_bill = request.form["nm"]
        size = request.form["rm"]
        smoker_value = request.form.get("smoker")
        gender_value = request.form.get("gender")
        week_day_value = request.form.get("week_day")
        day_time_value = request.form.get("day_time")
        results = {}
        results["total_bill"] = float(total_bill)
        results["smoker"] = smoker_value
```

```
        results["gender"] = gender_value
        results["week_day"] = week_day_value
        results["time"] = day_time_value
        results["number_of_people"] = int(size)
        prediction = predict(results)

        return render_template(
            "index.html",
            result=prediction,
            total_bill=total_bill,
            size=size,
            smoker_value=smoker_value,
            gender_value=gender_value,
            week_day_value=week_day_value,
            day_time_value=day_time_value,
        )

    return render_template("index.html")


if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=3500)
```

The page opens at port 3500. It shows the empty form first with all elements and populates them. Then, the user enters the data by selecting from or filling in the input cells with the POST request. The input elements are collected in the results dictionary and sent into the model as features. The model computes predictions based on the features in the predict module. Then all features and the prediction are rendered to the index.html to display on the web page.

The predict module resides in the /home/ubuntu/app/Waiter-Tips-Prediction/dags/app directory. The predict.py file contains the following code:

```
# Import libraries
import sys

sys.path.insert(
    1, "/home/ubuntu/app/Waiter-Tips-Prediction/dags/utils"
)
from typing import Any, Union

import aws_utils as aws
import mlflow
import numpy as np


def load_model() -> Any:

    """
```

```
    Loads the best model from the local folder or cloud bucket.
    """

    logged_model = aws.get_parameter("logged_model")
    loaded_model = mlflow.pyfunc.load_model(logged_model)

    return loaded_model


def transform_data(input_dict: dict[str, Union[int, float]]) -> list[Union[int, float]]:

    """
    Hot-encodes the data provided.
    """

    # Hot-encoding values
    gender = {"Female": 0, "Male": 1}
    smoker = {"Non-smoker": 0, "Smoker": 1}
    week_day = {"Thursday": 0, "Friday": 1, "Saturday": 2, "Sunday": 3}
    day_time = {"Lunch": 0, "Dinner": 1}

    # Replace the source dictionary values with the hot-encoding values
    input_dict["smoker"] = smoker[input_dict["smoker"]]
    input_dict["gender"] = gender[input_dict["gender"]]
    input_dict["week_day"] = week_day[input_dict["week_day"]]
    input_dict["time"] = day_time[input_dict["time"]]

    # Create features input to the model
    total_bill = input_dict["total_bill"]
    sex = input_dict["gender"]
    smoker = input_dict["smoker"]
    day = input_dict["week_day"]
    day_time = input_dict["time"]
    size = input_dict["number_of_people"]

    features = np.array([[total_bill, sex, smoker, day, day_time, size]])

    return features


def predict(input_dict: dict[str, Union[int, float]]) -> float:

    """
    Using the best model and features entered by user,
    predicts the target value.
    """

    # Load the model
```

```
    model = load_model()

    # Transform data
    features = transform_data(input_dict)

    # Make the prediction
    prediction = model.predict(features)[0]

    return f"{prediction:.2f}"
```

The app.py refers to the predict function in the predict module. The predict function, as said above, takes the results dictionary as input.

The predict first gets the model_uri we saved in the *AWS Parameter Store* as logged_model. It has a value like s3://s3b-tip-predictor/mlflow/1/eae9b2d38767486aa264cfb4253232e7/artifacts/models_mlflow/. Then using the model_uri, we request *MLFlow* to load our *Python* model. It is the best model we have recorded and staged for production.

The second task the predict function does is to transform the data entered by the user. It hot-encodes the features.

The final task is to make a prediction based on the transformed features. Once the app.py module gets the prediction results, it publishes them on the index.html page as in the following exhibit:

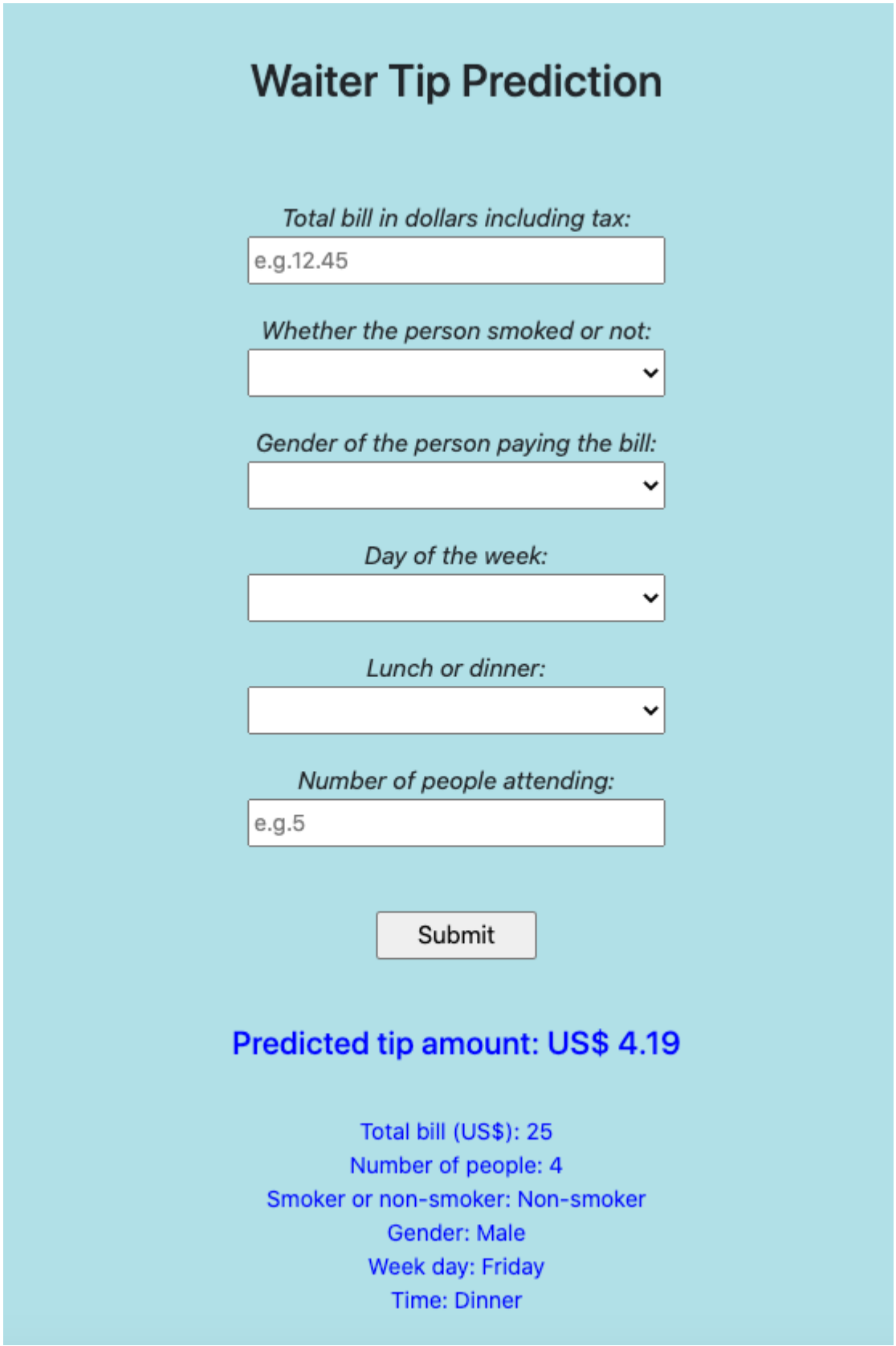



Exhibit-12: Publishing the Result on the Web Page (Image by Author)

Given that the total bill is $25, there are four people at the table, the person paying the bill is a non-smoker male, and it is a dinner on Friday, the predicted tip is $4.19!