Final Project Assignment
DADS 6005 : Data Streaming and Real-Time Analytics

Chayaphon Sornthananon, 6610422007

Semester 1/2024

Lecturer : Asst.Prof Ekarat Rattagan

Table of Content

Requirements

รันระบบ airflow สามวัน ดังนี้ วันที่ 21, 22, 23 ธค โดยในแต่ละวันจะกำหนดช่วงเวลาในการ predict BTC ตั้งแต่ 22:00 ~ 6:00 โมงเช้า โดย schedule ทุกๆ 5 นาที และคำนวณ mape ทุกๆ ชั่วโมง (หมายถึง len=12) โดย นศ จะส่ง mape.txt 3 files (ของช่วงวันเวลาที่ทดสอบ 3 วัน) + code 3 dags + requirement.txt และ files อื่นๆ ถ้ามีความแตกต่างจาก starter kit กำหนด

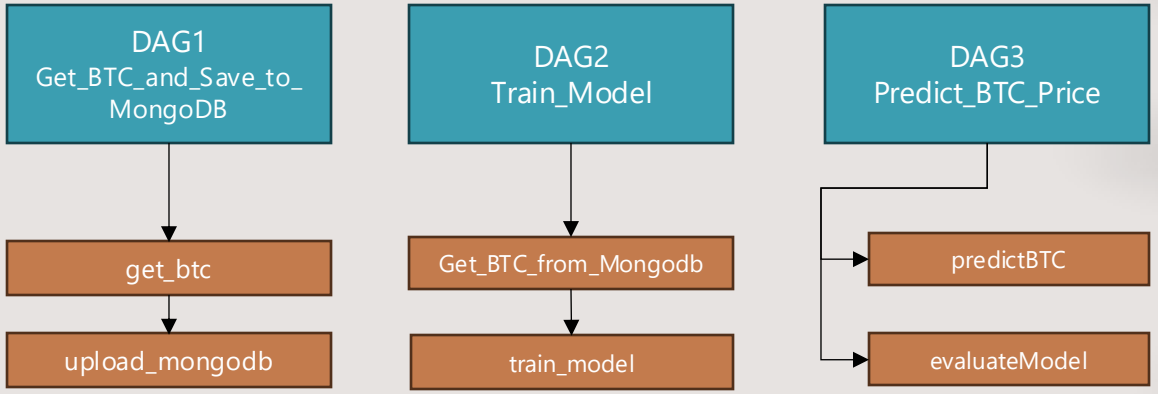
1.dataset ในการสร้าง train และ predict model จะมี column (X=closeTime) และ (Y=lastPrice)
2.นศ สามารถใช้ regression model ต่างๆ หรือ เทคนิคทางด้าน time series หรือ AI มาช่วยวิเคราะห์ได้

Airflow DAG Diagram

DAG

Task

Schedule

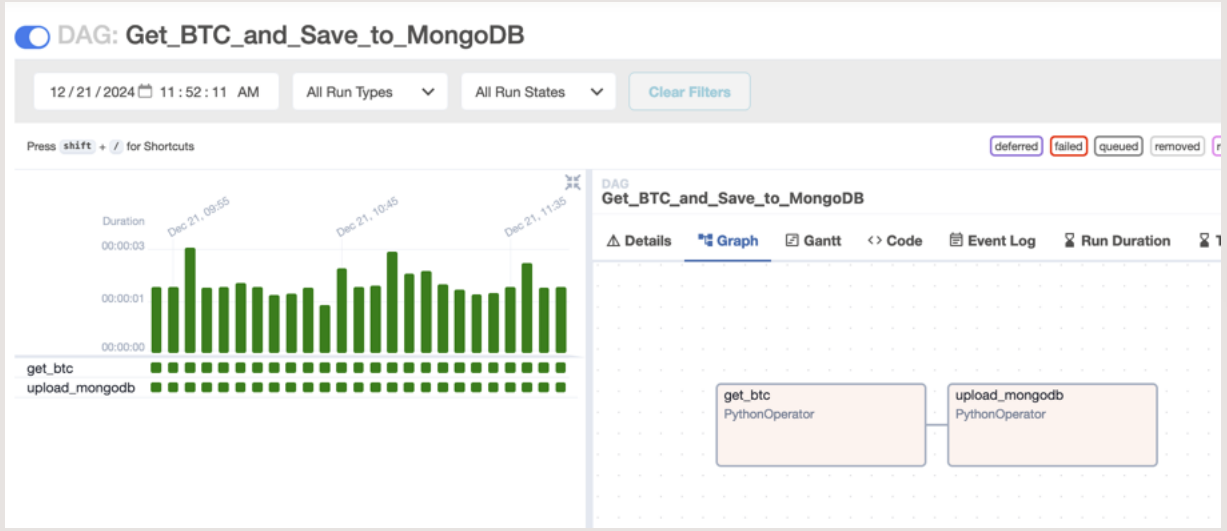


Schedule : Every 5 minutes
Start 2024-12-21 00:00 UTC7
End 2024-12-24 06:00 UTC7

Schedule : Every hour at minute 55
Start 2024-12-21 00:00 UTC7
End 2024-12-24 06:00 UTC7

Schedule : Every 5 minutes at 22:00 – 06:00 nextday from December date 21-23
Start 2024-12-21 21:00 UTC7
End 2024-12-24 06:00 UTC7

DAG 1 : Get_BTC_and_Save_to_MongoDB



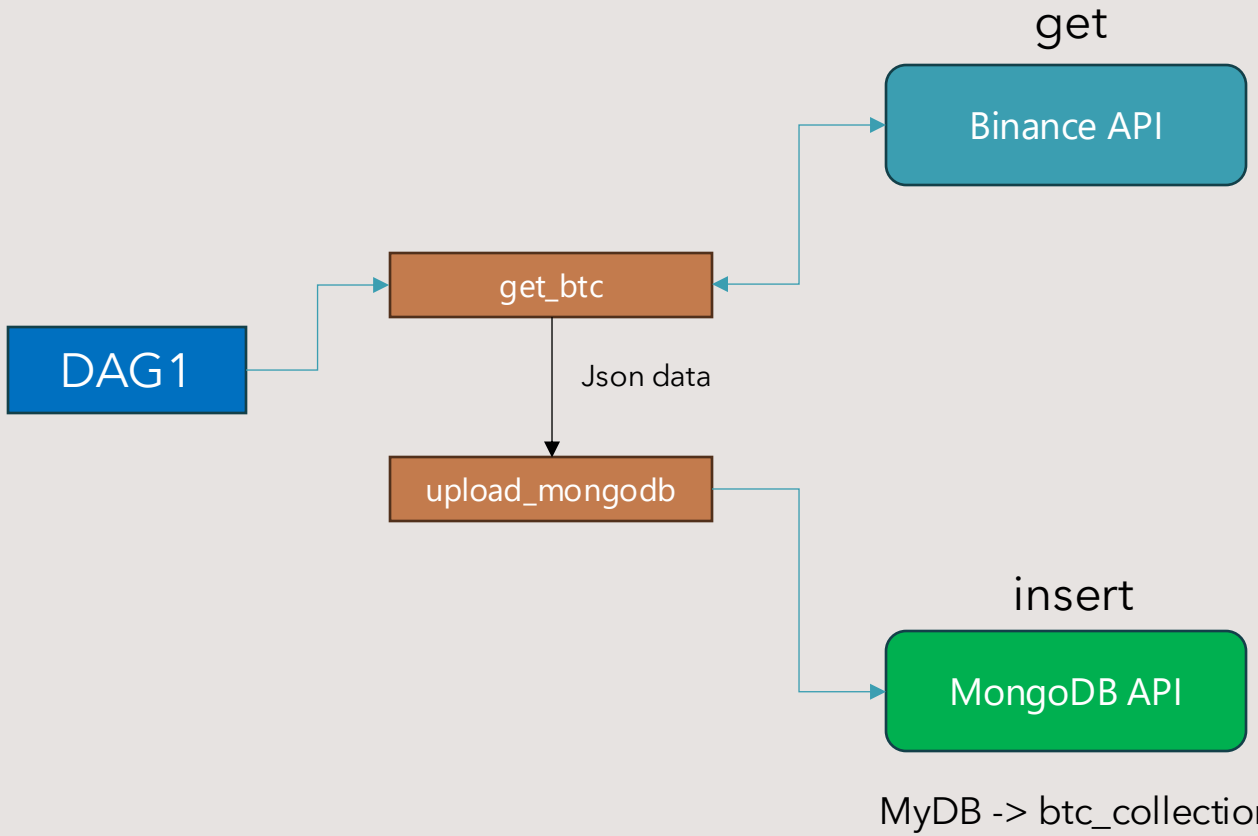
There are 2 task in DAG 1 as following :

- **Get_btc** : This task use https requests to get current BTCUSDT price from Binance API with windows size of 1 hours (average last 1 hour metrics) which contain data ('_id', 'symbol', 'openPrice', 'highPrice', 'lowPrice', 'lastPrice','volume', 'quoteVolume', 'openTime', 'closeTime', 'firstId', 'lastId','count') in json format.
- **Upload_mongodb** : This task insert the json data output from above task to MongoDB cloud (MyDB->btc_collection)

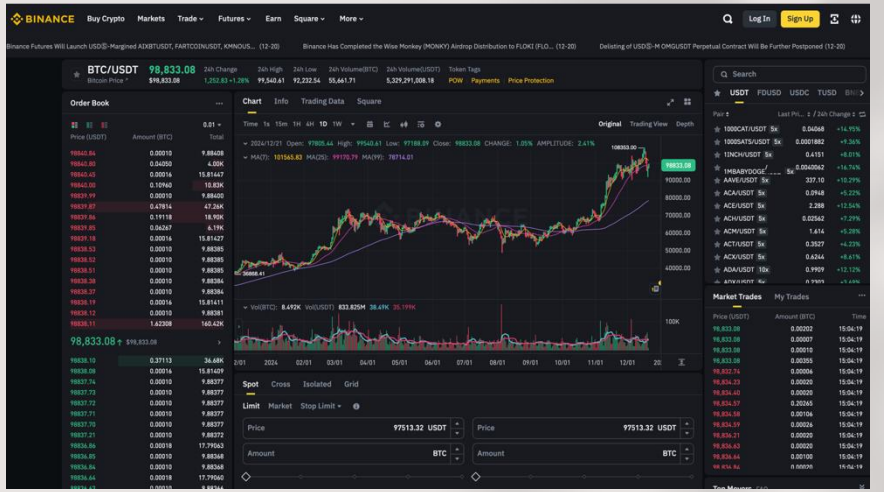
Schedule : Every 5 minutes. start 2024-12-21 00:00 utc7, end 2024-12-24 06:00 utc7

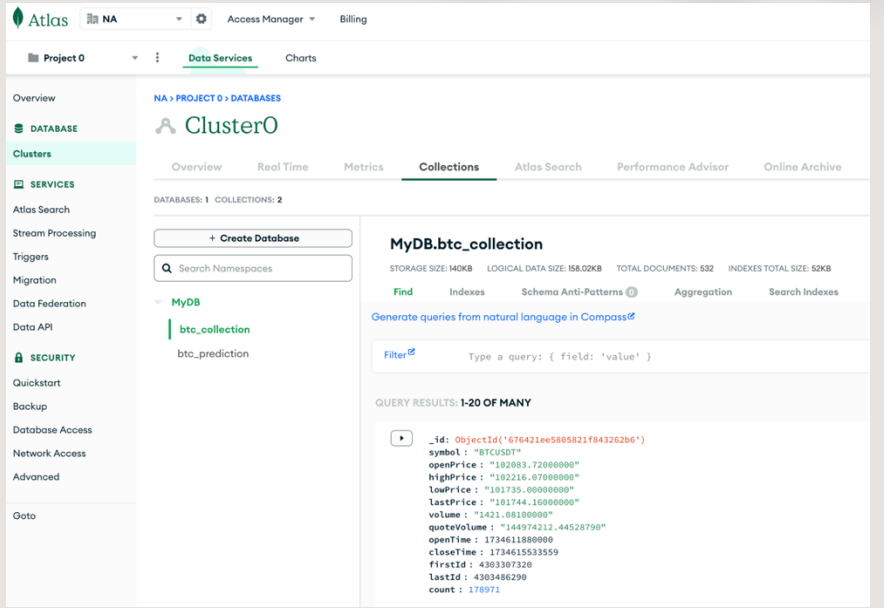
**More detail please check on code at dag1.py*

DAG 1 : Get_BTC_and_Save_to_MongoDB

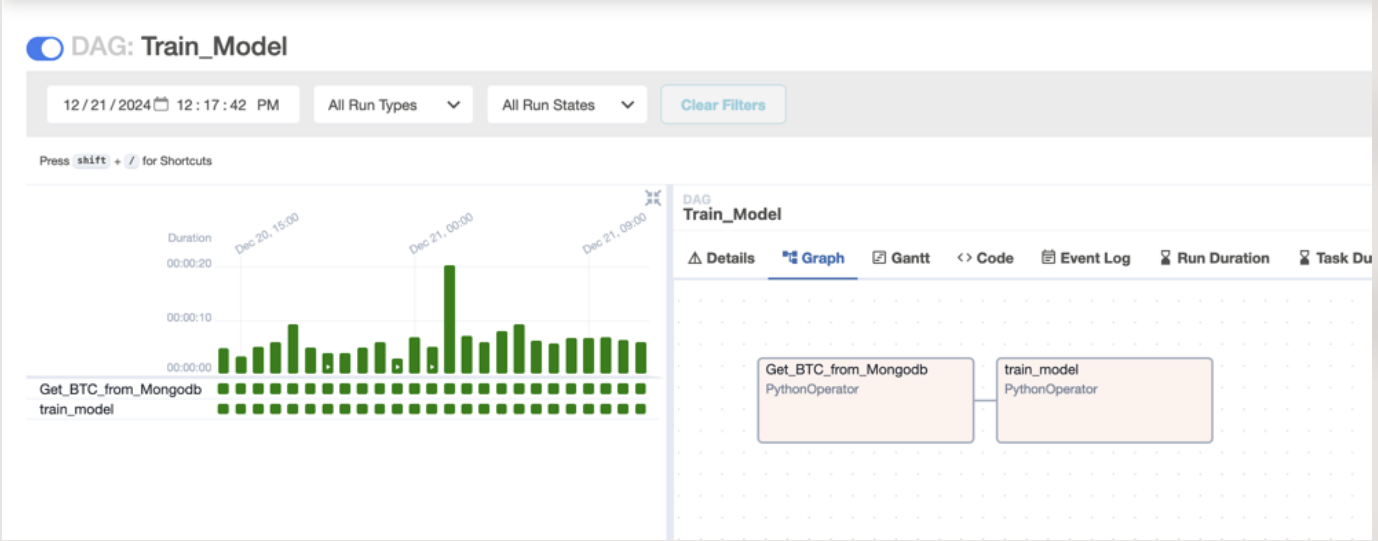


MyDB -> btc_collection





DAG 2 : Train_Model



There are 2 task in DAG 2 as following :

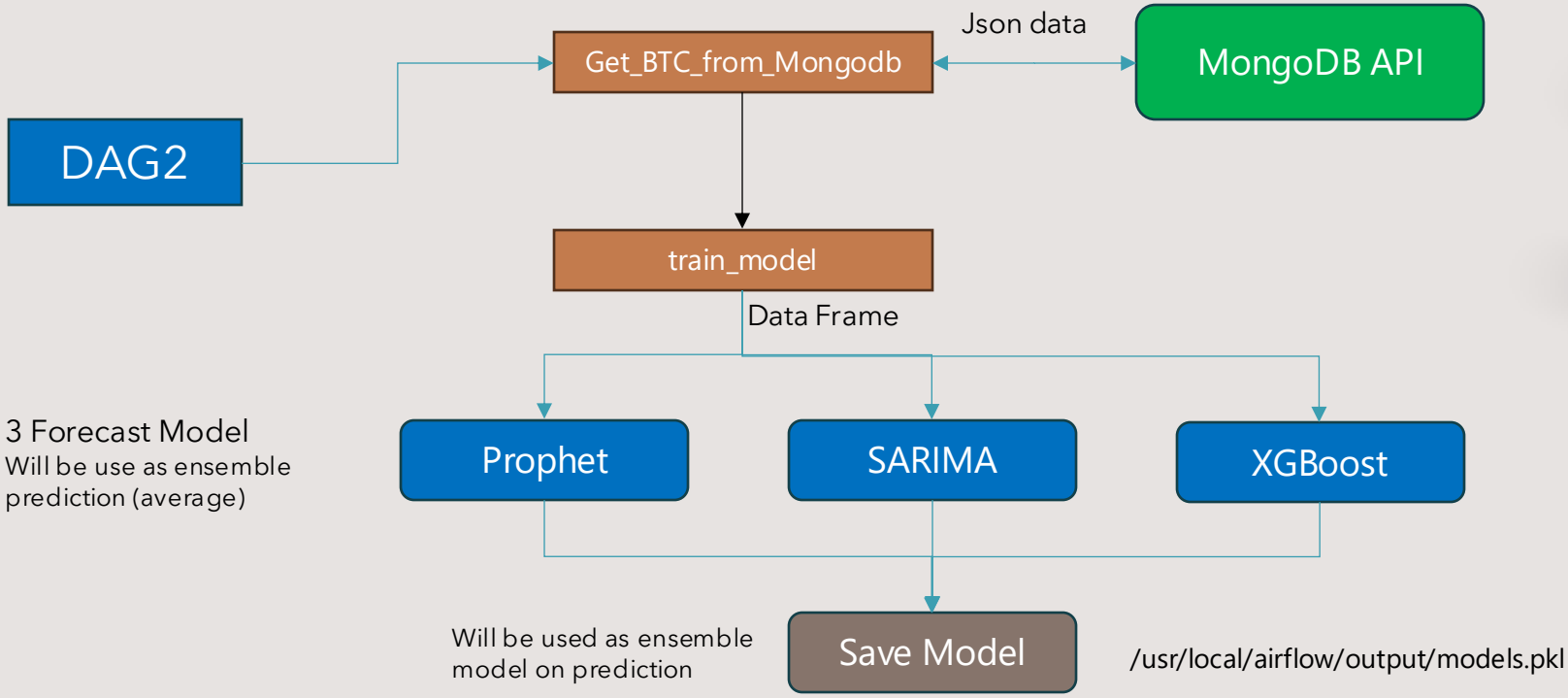
- **Get_BTC_from_Mongodb** : This task load data (last1day from current) from MongoDB cloud (MyDB->btc_collection).

- **Train_model** : use data output from above task (last 1day from current) with closeTime(X) and lastPrice(y) to train model.
  Then save model to "/usr/local/airflow/output/models.pkl"
  This project come with the idea of **ensemble model** of (Prophet, SARIMA and XGBoost) so we train all these 3 models in once.
  - **Prophet:** Suitable for handling seasonality and trend changes, modeled with multiplicative seasonality.
  - **SARIMA:** Incorporates both seasonal and non-seasonal components for autoregressive integrated moving average predictions.
  - **XGBoost:** A tree-based ensemble method optimized for regression tasks, trained with engineered time-series features.

Schedule : Every 1 hour at 55 minutes of each hour, e.g. 01:55, 02:55, 03:55 ... onward. start 2024-12-21 00:00 utc7, end 2024-12-24 06:00 utc7.

**More detail please check on code at dag2.py*

DAG 2 : Train_Model




```python
# Save all models
import pickle
models = {
    'prophet': model,
    'sarima': sarima_results,
    'xgb': xgb_model,
    'features': features,
    'last_timestamp': df['datetime'].max(),
    'last_prices': df['lastPrice'].tail(6).tolist()  # Save last 6 prices for lag features
}

modelFile = '/usr/local/airflow/output/models.pkl'
with open(modelFile, 'wb') as f:
    pickle.dump(models, f)
```

DAG 2 : Train_Model

Data Preparation

Load past 1 day to current data (288 reccords)
From MongoDB btc_collection

```python
def downloadFromMongo(ti, **kwargs):
    try:
        hook = MongoHook(mongo_conn_id='mongo_default')
        client = hook.get_conn()
        db = client.MyDB
        btc_collection = db.btc_collection
        print(f"Connected to MongoDB - {client.server_info()}")

        # Convert ObjectId to string
        def convert_object_id(data):
            if isinstance(data, list):
                return [convert_object_id(item) for item in data]
            elif isinstance(data, dict):
                return {key: convert_object_id(value) for key, value in data.items()}
            elif isinstance(data, ObjectId):
                return str(data)
            else:
                return data

        # Fetch and convert data
        # Calculate timestamp for 1 day ago
        one_day_ago = int((datetime.now() - timedelta(days=1)).timestamp() * 1000)
        #### Query to get only data with closeTime in the last 1 day ####
        query = {"closeTime": {"$gte": one_day_ago}}
        raw_data = list(btc_collection.find(query))
        converted_data = convert_object_id(raw_data)
        results = json.dumps(converted_data)
        print(results)

        ti.xcom_push(key="data", value=results)  # Use 'ti' to push to XCom

    except Exception as e:
        print(f"Error connecting to MongoDB -- {e}")
```

As requirement in forcasting use only clsosed time and lastPrice for features.
X = closedTime only
Y = lastPrice

```python
data = ti.xcom_pull(
    task_ids="Get_BTC_from_Mongodb",
    key="data"
)
df = pd.DataFrame(json.loads(data))

# prep data
df['datetime'] = pd.to_datetime(df['closeTime'], unit='ms')
df = df[~df['datetime'].duplicated(keep='first')]
df.set_index('datetime', inplace=True, drop=False)
df = df.sort_index()
df['lastPrice'] = pd.to_numeric(df['lastPrice'], errors='coerce')
df['lastPrice'] = df['lastPrice'].ffill() # Forward fill
```

1. Load past 1 day data (288 records; 24*12)
2. Add column datetime from ClosedTime(Unix Timestamp)
3. Datacleaning by remove duplicate by datetime(closedTime)
4. Set index as datetime and keep the column datetime as is.
5. Sorting index (datetime) ascending (*important in time series)
6. Ensure lastPrice is in numeric type
7. If there is null fill price with next row (*make sense in time serise like price in shorttime gap per each record.)

Features Engineering which will be use in XGBoost Regression

```python
# Feature Engineering for Time Series
df['lag1'] = df['lastPrice'].shift(1) # previous 1 price as features
df['lag2'] = df['lastPrice'].shift(2) # previous 2 price as features
df['lag3'] = df['lastPrice'].shift(3) # previous 3 price as features
df['rolling_mean_6'] = df['lastPrice'].rolling(window=6).mean() # moving average for 6 lastPrice
df['rolling_std_6'] = df['lastPrice'].rolling(window=6).std() # moving sd for 6 lastPrice
```

Enhance more feature from lastPrice

- Use previous lastPrice from 1, 2, 3 period.
- Use moving average and SD of lastPrice for last 6 period.

DAG 2 : Train_Model

Model Training (Prophet, SARIMA)

Prophet

X = datetime(closedTime)
Y = lastPrice

```python
from prophet import Prophet
# Prepare data for Prophet
prophet_df = pd.DataFrame()
prophet_df['ds'] = df['datetime']
prophet_df['y'] = df['lastPrice']

model = Prophet(
    changepoint_prior_scale=0.001,
    seasonality_prior_scale=0.1,
    seasonality_mode='additive',
    daily_seasonality=True,
    weekly_seasonality=True
)
model.fit(prophet_df)
```

Use iterative function to find the best parameter.

```
=== Best Prophet Parameters ===
Changepoint Prior Scale: 0.001
Seasonality Prior Scale: 1.0
Seasonality Mode: additive
Best RMSE: 621.72

Top 5 Parameter Combinations:
   changepoint_prior_scale  seasonality_prior_scale seasonality_mode  \
3                    0.001                      1.0         additive
7                    0.001                     10.0         additive
2                    0.001                      1.0   multiplicative
1                    0.001                      0.1         additive
5                    0.001                      5.0         additive
```

SARIMA

X = Index = datetime(closedTime)
Y = lastPrice

```python
from statsmodels.tsa.statespace.sarimax import SARIMAX
# Fit SARIMA model
sarima_model = SARIMAX(
    df['lastPrice'],
    order=(0, 1, 3),  # ARIMA parameters (p,d,q)
    seasonal_order=(0, 1, 1, 24),  # Seasonal parameters (P,D,Q,s)
)
sarima_results = sarima_model.fit()
```

Use auto_arima to find best param

```python
from pmdarima import auto_arima

# Fit auto_arima to find the best parameters
auto_model = auto_arima(
    df['lastPrice'],        # The original time series
    seasonal=True,          # Enable seasonal ARIMA
    m=24,                   # Seasonal period (24 for hourly data with daily seasonality)
    start_p=0, max_p=5,     # Range for
    start_q=0, max_q=5,     # Range for
    d=1,                    # Differen
    start_P=0, max_P=5,     # Range for
    start_Q=0, max_Q=5,     # Range for
    D=1,                    # Seasonal
    trace=True,             # Display
    error_action='ignore',  # Ignore e
    suppress_warnings=True, # Suppress
    stepwise=True           # Perform
)

# Display the best parameters
print(auto_model.summary())
```

```
ARIMA(0,1,4)(0,1,1)[24]             : AIC=3939.264, Time=1.75 sec
ARIMA(1,1,4)(0,1,1)[24]             : AIC=3940.153, Time=3.99 sec
ARIMA(0,1,3)(0,1,1)[24] intercept   : AIC=3940.583, Time=1.88 sec

Best model:  ARIMA(0,1,3)(0,1,1)[24]
Total fit time: 595.985 seconds
                                      SARIMAX Results
==========================================================================================
Dep. Variable:                                   y   No. Observations:                  288
Model:             SARIMAX(0, 1, 3)x(0, 1, [1], 24)   Log Likelihood               -1964.246
Date:                              Fri, 20 Dec 2024   AIC                           3938.493
Time:                                      22:56:00   BIC                           3956.354
Sample:                                           0   HQIC                          3945.671
                                                - 288
Covariance Type:                                opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ma.L1         -0.0085      0.034     -0.252      0.801      -0.075       0.058
ma.L2          0.0808      0.047      1.706      0.088      -0.012       0.174
ma.L3         -0.0892      0.054     -1.667      0.096      -0.194       0.016
ma.S.L24      -0.0905      0.010     -9.381      0.000      -0.109      -0.072
sigma2      1.818e+05   1.27e+04     14.306      0.000    1.57e+05    2.07e+05
==============================================================================
Ljung-Box (L1) (Q):                   3.35   Jarque-Bera (JB):                43.46
Prob(Q):                              0.07   Prob(JB):                         0.00
Heteroskedasticity (H):               0.82   Skew:                             0.01
Prob(H) (two-sided):                  0.36   Kurtosis:                         4.99
==========================================================================================
```

DAG 2 : Train_Model

Model Training (XGBboot : Regression)

```python
features = ['closeTime', 'lag1', 'lag2', 'lag3', 'rolling_mean_6', 'rolling_std_6']
X = df[features].dropna()
y = df['lastPrice'].loc[X.index]

# Train-test split
split_index = int(0.8 * len(X))
X_train = X[:split_index]
X_test = X[split_index:]
y_train = y[:split_index]
y_test = y[split_index:]

print(f"Training set size: {X_train.shape}, Test set size: {X_test.shape}")

# XGBoost parameters
params = {
    'objective': 'reg:squarederror',
    'learning_rate': 0.05,
    'max_depth': 3,
    'subsample': 0.9,
    'colsample_bytree': 0.7,
    'reg_alpha': 0.5,
    'reg_lambda': 2,
    'min_child_weight': 1,
    'eval_metric': 'rmse'
}

# Create DMatrix objects
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Set up evaluation list
evallist = [(dtrain, 'train'), (dtest, 'eval')]

# Train with early stopping
xgb_model = xgb.train(
    params,
    dtrain,
    num_boost_round=1000,
    evals=evallist,
    early_stopping_rounds=50,
    verbose_eval=True
)

# Calculate final metrics
train_pred = xgb_model.predict(dtrain)
test_pred = xgb_model.predict(dtest)

final_metrics = {
    'train_rmse': float(np.sqrt(mean_squared_error(y_train, train_pred))),
    'test_rmse': float(np.sqrt(mean_squared_error(y_test, test_pred))),
    'best_iteration': xgb_model.best_iteration,
    'feature_importance': dict(zip(features, [float(v) for v in xgb_model.get_score().values()])),
    'params': params
}
```

X = closedTime, lag1, lag2, lag3, rolling_mean_6, rolling_std6 as mentioned in previous data prep
y = lastPrice

model is performing **regression** and uses the **squared error loss** (L2 loss) as the optimization function.

Use RMSE as evaluation matrix

Training with 1000 iteration with early stopping (50), if there is no improvement in 50 iteration then stop before reaching 1000 iteration to save time

Use GridSearchCV to find best parameter

```python
# Create DMatrix objects
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Set up evaluation list
evallist = [(dtrain, 'train'), (dtest, 'eval')]

# Train with early stopping
xgb_model = xgb.train(
    params,
    dtrain,
    num_boost_round=1000,
    evals=evallist,
    early_stopping_rounds=50,
    verbose_eval=True
)

# Calculate final metrics
train_pred = xgb_model.predict(dtrain)
test_pred = xgb_model.predict(dtest)

final_metrics = {
    'train_rmse': float(np.sqrt(mean_squared_error(y_train, train_pred))),
    'test_rmse': float(np.sqrt(mean_squared_error(y_test, test_pred))),
    'best_iteration': xgb_model.best_iteration,
    'feature_importance': dict(zip(features, [float(v) for v in xgb_model.get_score().values()])),
    'params': params
}

print(f"Final Metrics: {json.dumps(final_metrics, indent=2)}")
```

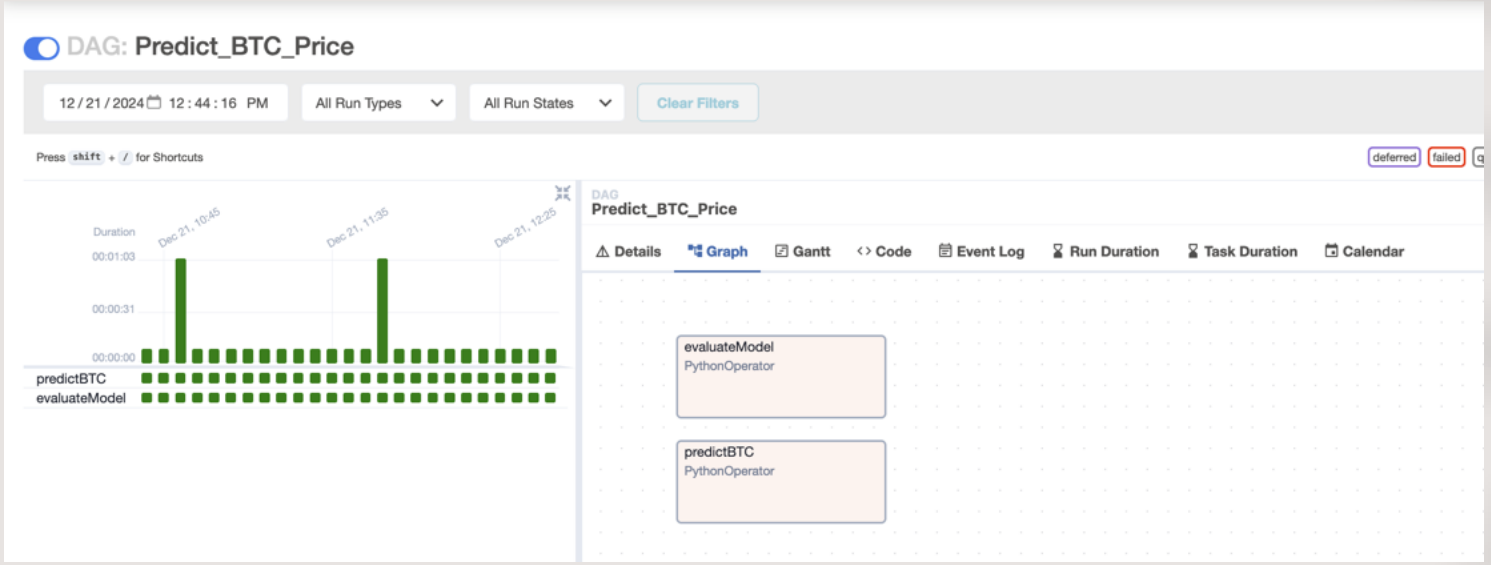
```
[CV] END colsample_bytree=0.9, learning_rate=0.1, max_depth=7, min_child_weight=5, reg_alpha=2, reg_lambda=3, subsample=0.9; total time=   0.0s
[CV] END colsample_bytree=0.9, learning_rate=0.1, max_depth=7, min_child_weight=5, reg_alpha=2, reg_lambda=3, subsample=0.9; total time=   0.0s
Best Parameters: {'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 3, 'min_child_weight': 1, 'reg_alpha': 0.5, 'reg_lambda': 2, 'subsample': 0.9}
Best RMSE: 332.97639318439826
```

```python
# Save all models
import pickle
models = {
    'prophet': model,
    'sarima': sarima_results,
    'xgb': xgb_model,
    'features': features,
    'last_timestamp': df['datetime'].max(),
    'last_prices': df['lastPrice'].tail(6).tolist()  # Save last 6 prices for lag features
}

modelFile = '/usr/local/airflow/output/models.pkl'
with open(modelFile, 'wb') as f:
    pickle.dump(models, f)
```

Save 3 models in .pkl

DAG 3 : Predict_BTC_Price



There are 2 independent task in DAG 3 as following :

- **predictBTC** : This task execute only every interval (in this case interval = 12 runs) has been met. So, each run in every 5 minutes means 12 * 5minutes = 1hour. It predicts using the saved model from dag2 output and **forecast BTC price in next 12-time interval**, e.g. if task executed at 10:55 then it will forecast BTC price for 11:00, 11:05, 11:10,… 11:55 and then save(overwrite) the latest output to **"/usr/local/airflow/output/out.csv"** as well as append it to the historical to **/usr/local/airflow/output/prediction_history.csv** for later analysis.

- **EvaluateModel** : use data output from above task "/usr/local/airflow/output/out.csv" (use ensemble predicted result) to calculate MAPE with actual prices.
  - Every run (5 minutes) it will load current lastPrice from Binance API as y-actual and compare error with y-hat (ensemble predicted result) and add into global variable list.
  - Every interval has been met (12 runs) it will calculate MAPE(Mean Absolute Percentage Error) from last 12 absolute error percentage and save to f'/usr/local/airflow/output/mape_{str_date}.txt'

Schedule : Every 5 minutes from 22:00-06:00 between december21-24. start 2024-12-21 00:00 utc7, end 2024-12-24 06:00 utc7

**More detail please check on code at dag3.py*

DAG 3 : Predict_BTC_Price

Load Model

/usr/local/airflow/output/models.pkl

Every T=12
(1 hour)

predictBTC

output

MongoDB

MyDB.btc_prediction

CSV

/usr/local/airflow/output/out.csv

```
# Ensemble predictions (simple average)
final_predictions = (prophet_predictions + sarima_predictions + xgb_predictions) / 3
# ---------------------------------- #
####################################

# Create DataFrame with predictions
prediction_df = pd.DataFrame({
    'createdTime' : current_time,
    'closeTime': pd.date_range(start=predicted_time, periods=interval, freq=freq_),
    'predicted': final_predictions,
    'prophet_pred': prophet_predictions,
    'sarima_pred': sarima_predictions,
    'xgb_pred': xgb_predictions
})

# save latest 12 predictions to out.csv
file_path1 = '/usr/local/airflow/output/out.csv'
prediction_df.to_csv(file_path1, index=False)
```

Every 5 minutes

DAG3

Binance API

Every T
(5minutes)

input

evaluateModel

output

hourly

MAPE txt

Keep APE each run in global variable once reach interval time then calculate MAPE and save output to txt

DAG 3 : Predict_BTC_Price

predictBTC : code inference (ensemble average result)

Inference Prophet forecast

Inference SARIMA forecast

Inference XGBoost forecast

```python
# predict ofr next 12 times starting at next 5 minutes.
# this will be use in model due to model was train with utc time
predicted_time = datetime.utcnow() +  timedelta(minutes=5) + timedelta(seconds=1)
# this will be use for display time in out.csv in thai time
predicted_time_th = current_time_th +  timedelta(minutes=5) + timedelta(seconds=1)

# This will generate forecast time slot starting from predicted_time with period interval by freq_ (future closeTime)
forecast_closeTime = pd.date_range(start=predicted_time, periods=interval, freq=freq_)

# Prophet Prediction for next interval period
future_dates = pd.DataFrame({'ds': forecast_closeTime})
prophet_forecast = models['prophet'].predict(future_dates)
prophet_predictions = prophet_forecast['yhat'].values
# ------------------------------------ #

# SARIMA Prediction for next interval period
# sarima_predictions = models['sarima'].forecast(steps=interval)
sarima_predictions = models['sarima'].forecast(steps=interval, index=forecast_closeTime)
# ------------------------------------ #

# XGBoost Prediction for next interval period
# Prepare features for XGBoost
future_features = pd.DataFrame()
future_features['closeTime'] = forecast_closeTime
future_features['closeTime'] = future_features['closeTime'].astype('int64') // 10**6 # turn it to timestamp ms due to model was train by this format
future_features['lag1'] = models['last_prices'][-1]
future_features['lag2'] = models['last_prices'][-2]
future_features['lag3'] = models['last_prices'][-3]
future_features['rolling_mean_6'] = np.mean(models['last_prices'])
future_features['rolling_std_6'] = np.std(models['last_prices'])

# Convert future_features to DMatrix
dmatrix_future_features = xgb.DMatrix(future_features[models['features']])
# Predict using the XGBoost model
xgb_predictions = models['xgb'].predict(dmatrix_future_features)
# ------------------------------------ #

# Ensemble predictions (simple average)
final_predictions = (prophet_predictions + sarima_predictions + xgb_predictions) / 3
# ------------------------------------ #

# Create DataFrame with predictions
prediction_df = pd.DataFrame({
    'createdTime' : current_time_th,
    'closeTime': pd.date_range(start=predicted_time_th, periods=interval, freq=freq_),
    'predicted': final_predictions,
    'prophet_pred': prophet_predictions,
    'sarima_pred': sarima_predictions,
    'xgb_pred': xgb_predictions
})

# save latest 12 predictions to out.csv
file_path1 = '/usr/local/airflow/output/out.csv'
prediction_df.to_csv(file_path1, index=False)
```

predicted_time as datetime utc used in training model so we will inference model with the same data structure format.
+ 5 minutes because the forecast price should start on next 5 minutes btc price onward.
+ offset 1 second for task starting cost.

Forecast_closeTime will generate series of datetime in each (interval=12 ; records) and each interval is increasing in freq=5 minutes
Starting from **predicted_time**
e.g. starting time = 2024-12-21 22:05:00 then it will generate 2024-12-21 22:05:00, 2024-12-21 22:10:00, 2024-12-21 22:15:00 … 2024-12-21 22:55:00.

Ensemble result with average result of 3 models

Create Data frame result and save to out.csv

Example of Output

out.csv

- write every 1 hour

createdTime	closeTime	predicted	prophet_pred	sarima_pred	xgb_pred
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:05:02.328255+07:00	95480.67276	95219.5978	95462.08454	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:10:02.328255+07:00	95507.3187	95236.55105	95525.06911	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:15:02.328255+07:00	95468.04296	95253.84045	95389.95248	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:20:02.328255+07:00	95549.81703	95271.32149	95617.79367	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:25:02.328255+07:00	95497.2787	95288.84884	95442.65131	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:30:02.328255+07:00	95512.43296	95306.27737	95470.68559	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:35:02.328255+07:00	95529.39509	95323.46316	95504.38619	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:40:02.328255+07:00	95495.47118	95340.26451	95385.81309	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:45:02.328255+07:00	95481.42088	95356.5429	95327.3838	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:50:02.328255+07:00	95485.46549	95372.16391	95323.89661	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 06:55:02.328255+07:00	95481.93737	95386.99816	95298.478	95760.336
2024-12-23 06:00:01.328255+07:00	2024-12-23 07:00:02.328255+07:00	95379.66291	95400.92213	94977.73066	95760.336

Use as predicted price each T(5 minutes) to calculate APE and then MAPE at the end of T(12)
Remark: predicted =(prophet_pred+sarima_pred+xgb_pred)/3

mape_yyyy-mm-dd.txt

- append every 1 hour

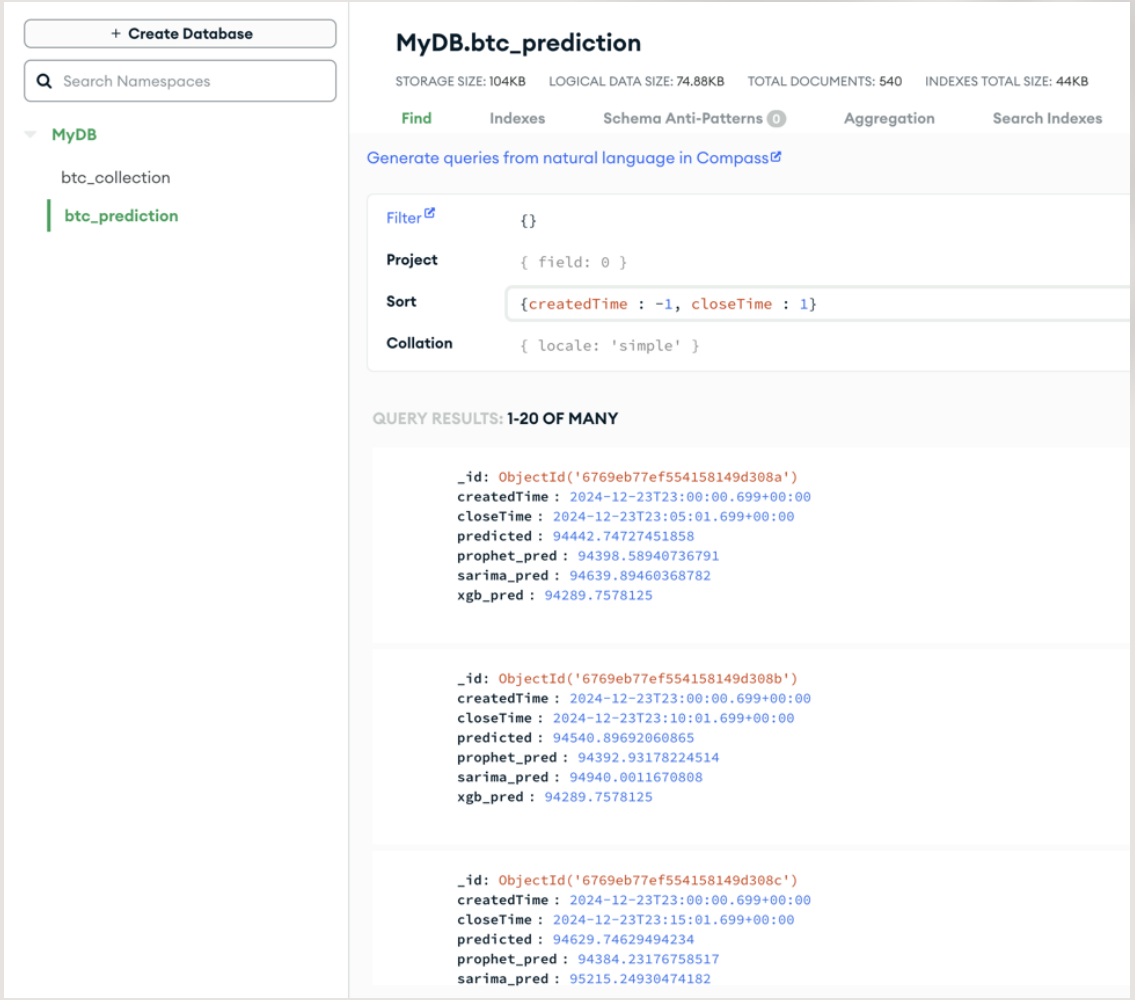
mape_2024-12-23.txt

```
time, mape
2024-12-23 23:00:01.508251+07:00, 0.8757151772568086
2024-12-24 00:00:00.501407+07:00, 0.6924791788102762
2024-12-24 01:00:00.993524+07:00, 0.38809660191634643
2024-12-24 02:00:00.942024+07:00, 0.5397986858375143
2024-12-24 03:00:00.768641+07:00, 0.21193403650303966
2024-12-24 04:00:00.834085+07:00, 0.8157276056855576
2024-12-24 05:00:01.524880+07:00, 0.7295407806492584
2024-12-24 06:00:00.869186+07:00, 0.834218185192831
```

Use for calculate MAPE(mean absolute percentage error) from APE of each 5 minutes past hour (12T, 5 minutes each)

Mongodb btc_prediction

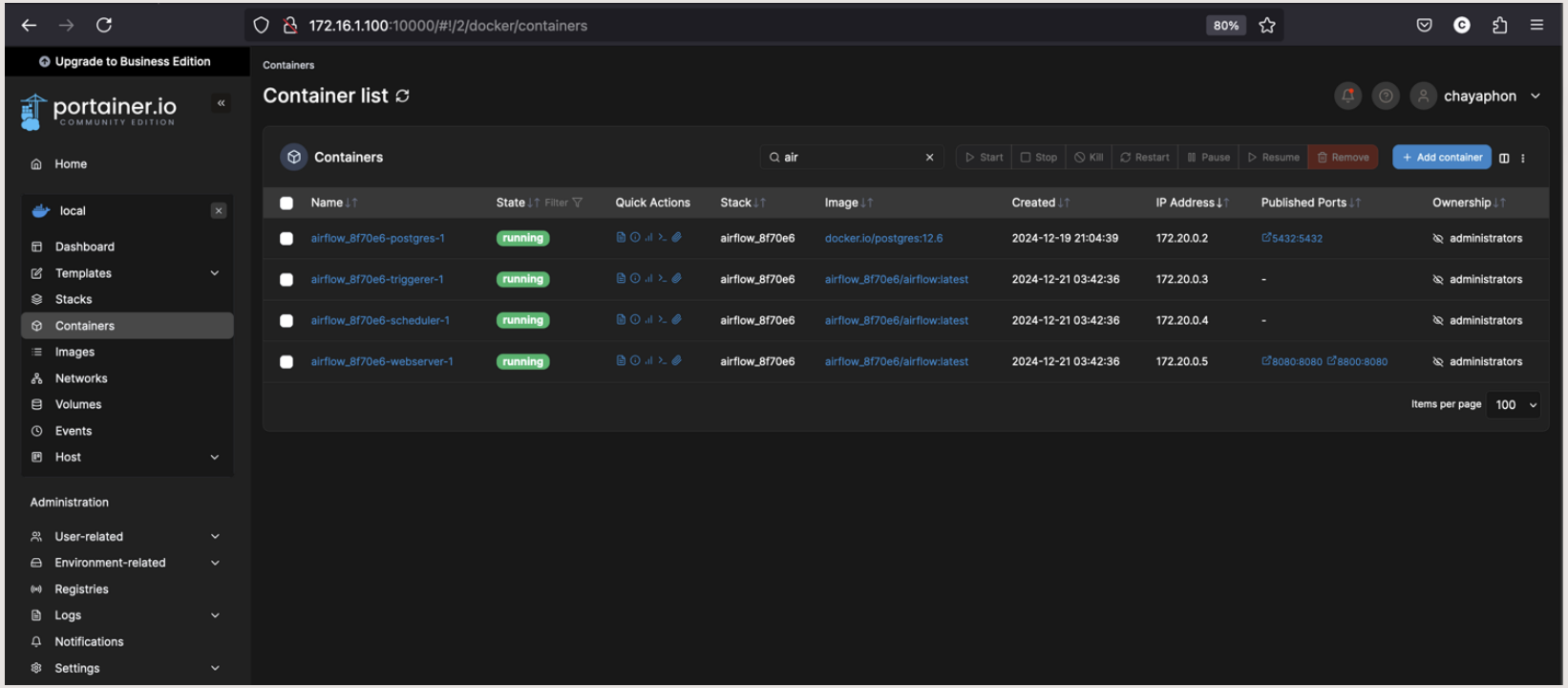
- Insert the prediction of every 1 hours in advance into mongo for later analysis.



Appendix

Docker

In this project we use docker technology to help us running airflow and related services.



docker-compose.override.yml

Orverride the standard config of astro docker

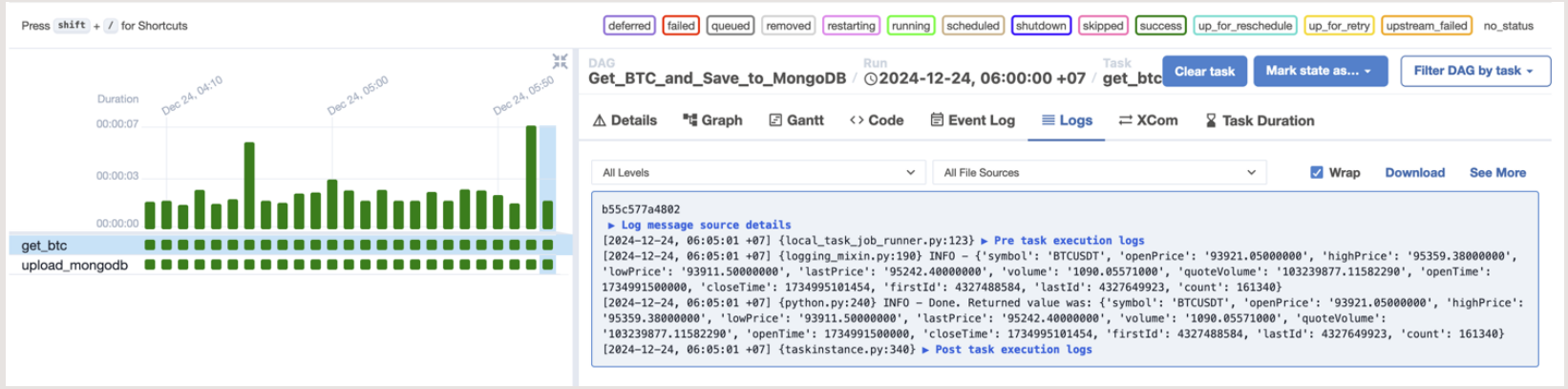
```yaml
version: '3'
services:
  scheduler:
    volumes:
      - ./output:/usr/local/airflow/output
  webserver:
    # Map host port 8800 to container port 8080,Because my docker server has already used port 8080 for other service.
    ports:
      - "8800:8080"
    # below command allow me to access it from client pc (I hosted docker in server and access it from client)
    command: >
      airflow webserver --port 8080 --host 0.0.0.0
```

I have created local directory "**output**" which map to docker airflow "**usr/local/airflow/output**" where I use to dump output of task inside docker to persistent in local machine.

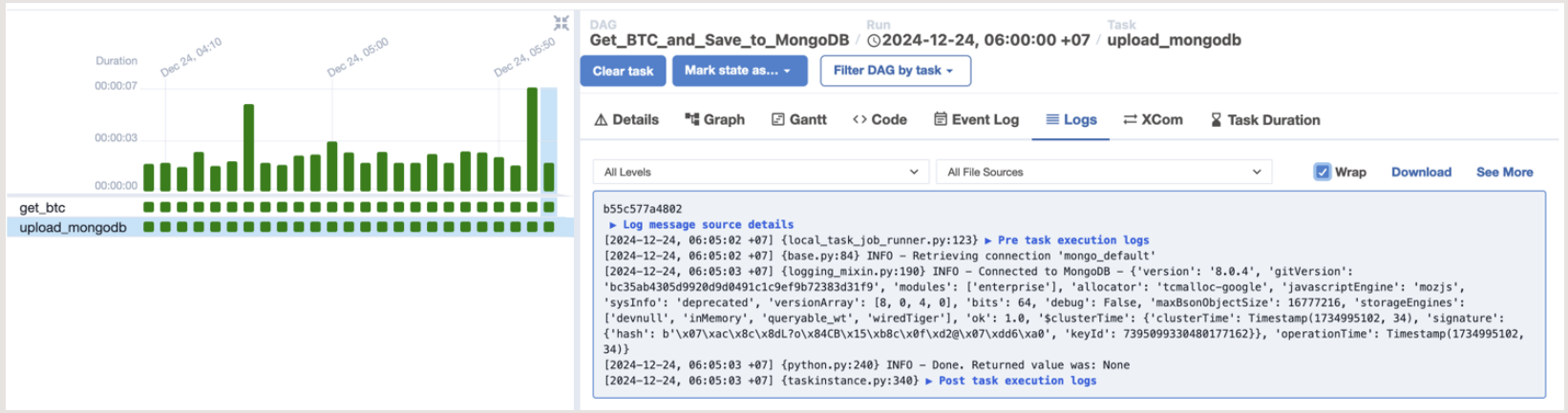
Some of Task Output Screen Shot

Get_BTC_and_Save_to_MongoDB

get_btc



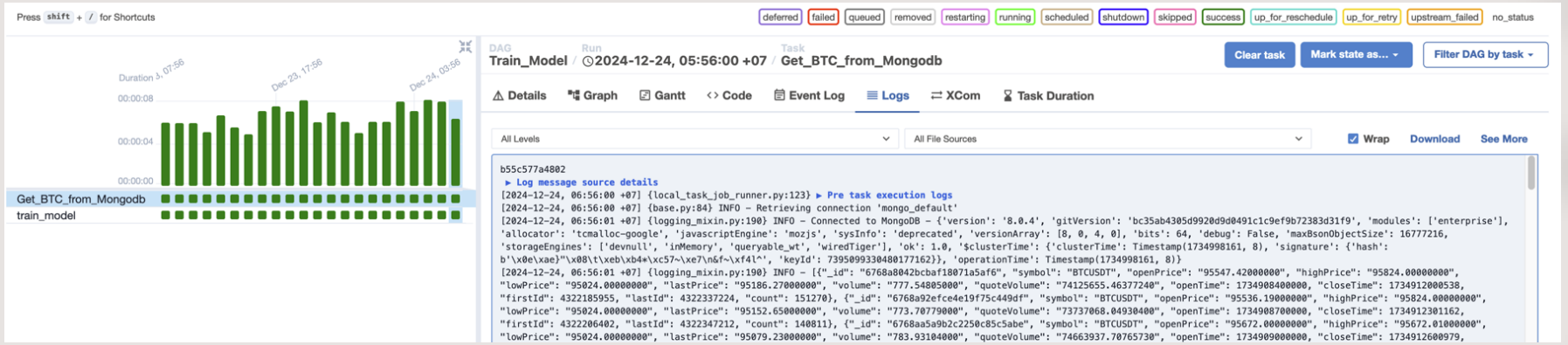
upload_mongodb



Some of Task Output Screen Shot

Tran_Model

Get_BTC_from_Mongodb. (last 1 day data 288 records)

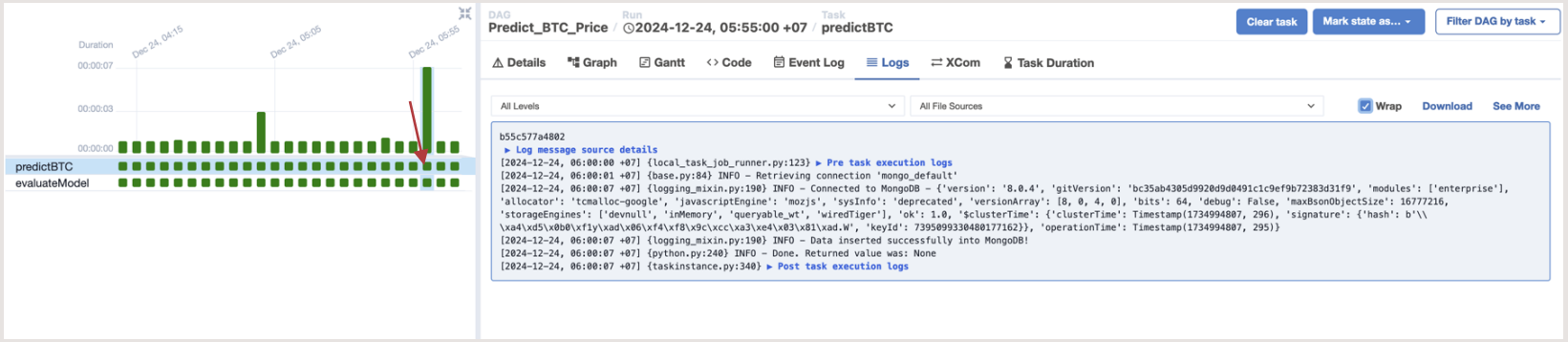


Train_model

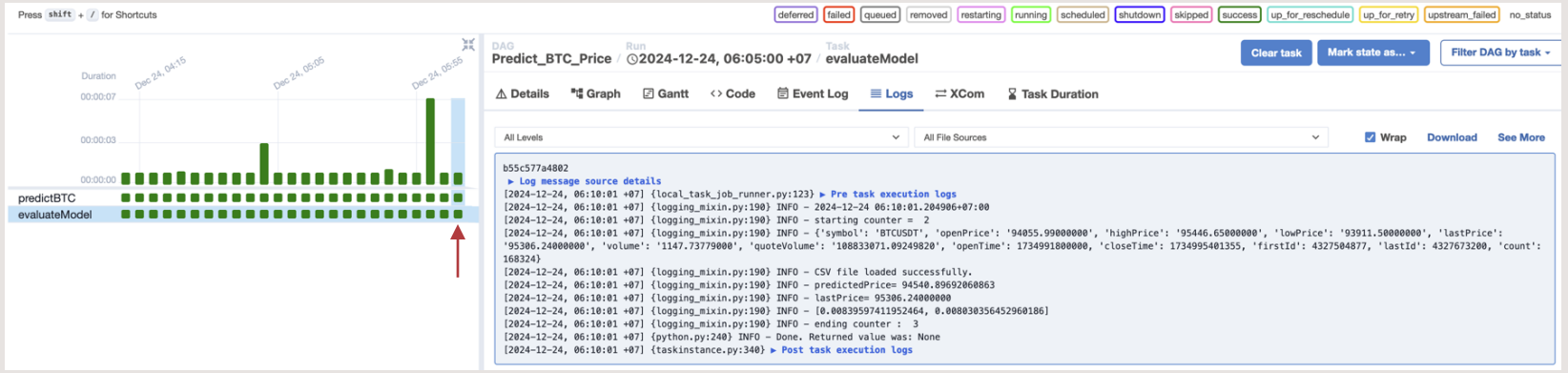
Some of Task Output Screen Shot

Tran_Model

predictBTC : write predicted price in next 12T (5 minutes each) to out.csv and insert data to MongoDB

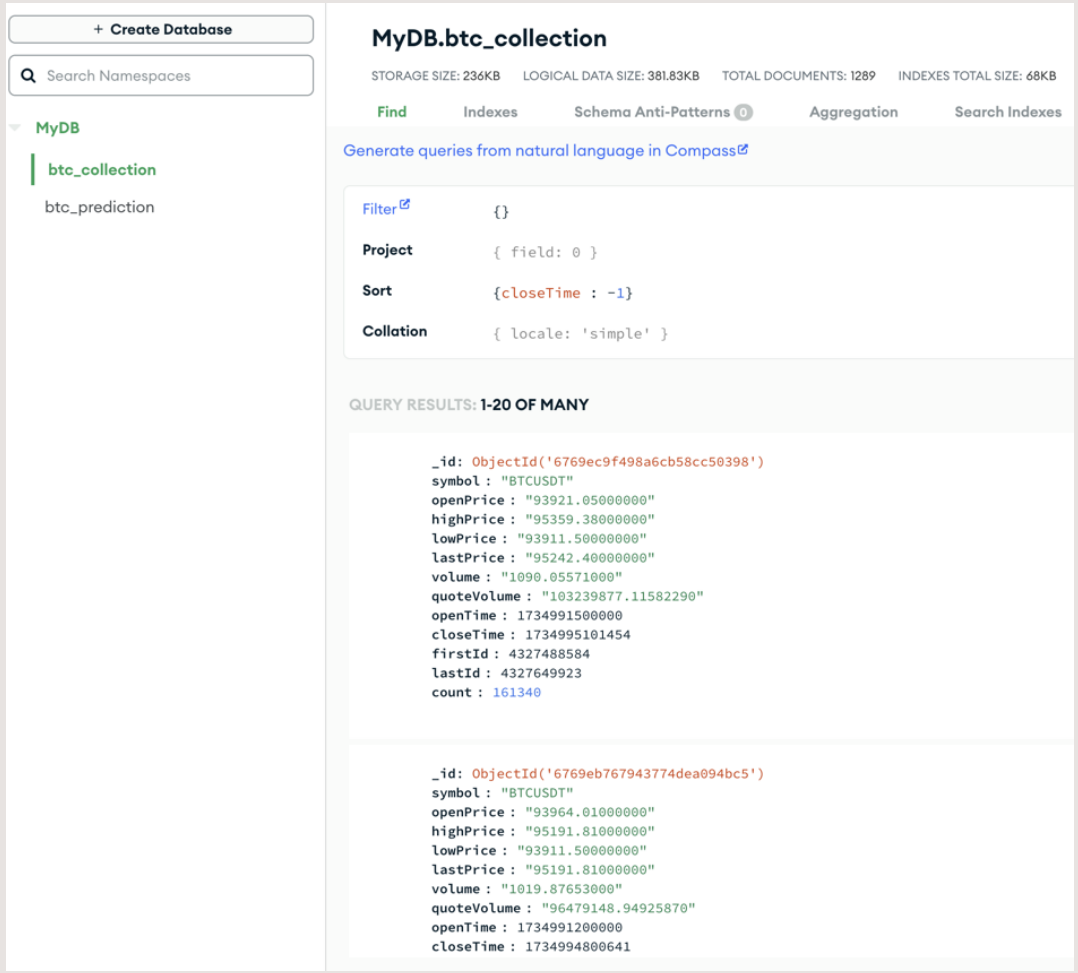


evaluateModel : calculate error (predict-actual)/actual for each T (5 minutes)

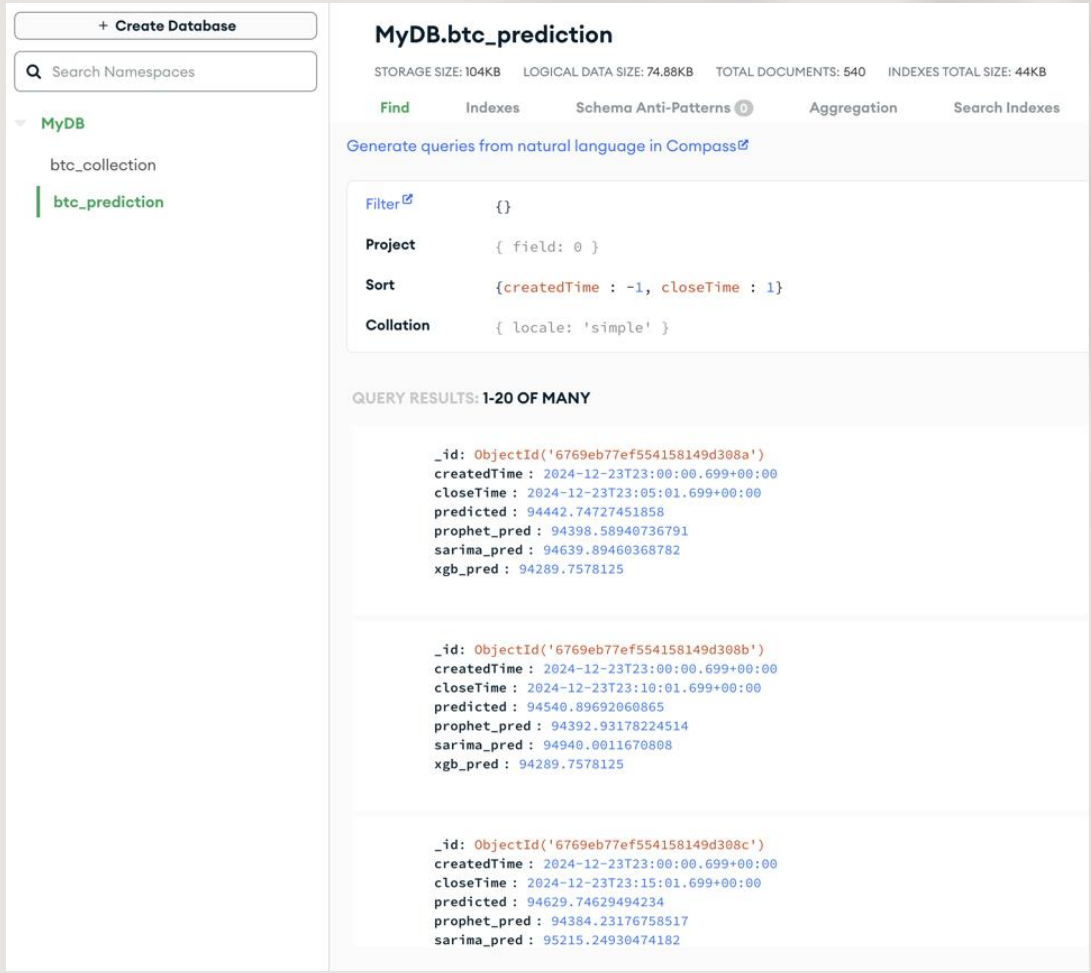


Mongo DB

btc_collection



btc_prediction



Code Reference

https://github.com/chayaphon/realtime_airflow-btcPrediction