SOURCE CODE

```
import numpy as np
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import math
# Load dataset
data = pd.read_csv('/content/cloud_workload_dataset.csv',
parse_dates=['timestamp'], index_col='timestamp')
# Preprocessing
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data.values)
# Split data into training and testing sets
train_size = int(len(scaled_data) * 0.8)
train_data = scaled_data[:train_size]
test_data = scaled_data[train_size:]
# ARIMA Model
def train_arima(train_series, order=(5, 1, 0)):
 model = ARIMA(train_series, order=order)
 model_fit = model.fit()
 return model_fit
arima_predictions = []
```

```
for i in range(len(test_data)):
 train_series = scaled_data[:train_size + i].flatten()
 arima_model = train_arima(train_series)
 forecast = arima_model.forecast(steps=1)
 arima_predictions.append(forecast[0])
arima_predictions = np.array(arima_predictions).reshape(-1, 1) # Ensure correct
shape
# LSTM Model
def create_lstm_model(input_shape):
 model = Sequential()
 model.add(LSTM(50, return_sequences=True, input_shape=input_shape))
 model.add(LSTM(50, return_sequences=False))
 model.add(Dense(25))
 model.add(Dense(1))
 model.compile(optimizer='adam', loss='mean_squared_error')
 return model
def prepare_data_for_lstm(data, time_steps=30):
 x, y = [], []
 for i in range(time_steps, len(data)):
 x.append(data[i-time_steps:i, 0])
 y.append(data[i, 0])
 return np.array(x), np.array(y)
# Define time_steps
time_steps = min(30, len(test_data) - 1)
x_train, y_train = prepare_data_for_lstm(train_data, time_steps)
x_test, y_test = prepare_data_for_lstm(test_data, time_steps)
# Reshape for LSTM
```

```
x_train = x_train.reshape((x_train.shape[0], x_train.shape[1], 1))
x_test = x_test.reshape((x_test.shape[0], x_test.shape[1], 1))
# Train LSTM Model
lstm_model = create_lstm_model((x_train.shape[1], 1))
lstm_model.fit(x_train, y_train, batch_size=32, epochs=20, verbose=1)
lstm_predictions = lstm_model.predict(x_test)
# Ensure ARIMA predictions align with LSTM predictions
arima_predictions = arima_predictions[-len(lstm_predictions):] # Trim to match
length
# Combine ARIMA and LSTM predictions
combined_predictions = (arima_predictions.flatten() + lstm_predictions.flatten()) /
2
# Reverse scaling
arima_predictions_rescaled = scaler.inverse_transform(arima_predictions)
lstm_predictions_rescaled = scaler.inverse_transform(lstm_predictions)
combined_predictions_rescaled =
scaler.inverse_transform(combined_predictions.reshape(-1, 1))
y_test_rescaled = scaler.inverse_transform(y_test.reshape(-1, 1))
# Performance metrics
def evaluate_model(true_values, predicted_values):
 mae = mean_absolute_error(true_values, predicted_values)
 rmse = math.sqrt(mean_squared_error(true_values, predicted_values))
 r2 = r2_score(true_values, predicted_values)
 return mae, rmse, r2
arima_mae, arima_rmse, arima_r2 = evaluate_model(y_test_rescaled,
arima_predictions_rescaled)
lstm_mae, lstm_rmse, lstm_r2 = evaluate_model(y_test_rescaled,
```

```
lstm_predictions_rescaled)
combined_mae, combined_rmse, combined_r2 = evaluate_model(y_test_rescaled,
combined_predictions_rescaled)
print("ARIMA Performance: MAE =", arima_mae, "RMSE =", arima_rmse, "R2
=", arima_r2)
print("LSTM Performance: MAE =", lstm_mae, "RMSE =", lstm_rmse, "R2 =",
lstm_r2)
print("Hybrid Performance: MAE =", combined_mae, "RMSE =", combined_rmse,
"R2 =", combined_r2)
import pandas as pd
import matplotlib.pyplot as plt
# Load the data
# file_path = "workload_data.xlsx" # Update with the correct file path if needed
file_path = "/content/cloud_workload_dataset.csv" # Changed to load the csv file
that is available in /content/
data = pd.read_csv(file_path) # Changed to read_csv to read the CSV file
# Convert timestamp to datetime format
data['timestamp'] = pd.to_datetime(data['timestamp'])
# Plot the workload over time
plt.figure(figsize=(12, 6))
plt.plot(data['timestamp'], data['workload'], marker='o', linestyle='-',
label="Workload", color='blue')
# Customize the plot
plt.title("Workload Over Time", fontsize=16)
plt.xlabel("Timestamp", fontsize=14)
plt.ylabel("Workload", fontsize=14)
plt.grid(visible=True, linestyle="--", alpha=0.6)
```

```
plt.xticks(rotation=45)
plt.tight_layout()
plt.legend()
# Save the graph
plt.savefig("workload_plot.png", dpi=300)
plt.show()
import matplotlib.pyplot as plt
import numpy as np
# Example data (replace with real values)
# Actual values (True values from test set)
y_actual = [55, 57, 54, 53, 58, 60, 63, 62, 61, 65, 70, 75, 78, 80, 76, 74, 72, 68, 64,
63]
# Predictions from ARIMA, LSTM, and Hybrid models (replace with actual model
predictions)
y_pred_arima = [54, 56, 53, 52, 57, 59, 62, 61, 60, 64, 69, 74, 77, 79, 75, 73, 71,
67, 63, 62]
y_pred_lstm = [55, 57, 54, 53, 58, 60, 62, 62, 61, 64, 69, 74, 77, 79, 76, 74, 72, 68,
65, 64]
y_pred_hybrid = [34, 33, 36, 40, 41, 40, 43, 40,39, 38, 37, 36, ]
# Create the plot
plt.figure(figsize=(10, 6))
# Plot actual values
plt.plot(y_actual, label='Actual Values', color='black', linestyle='-', marker='o')
# Plot ARIMA predictions
plt.plot(y_pred_arima, label='ARIMA Predictions', color='blue', linestyle='--',
marker='x')
# Plot LSTM predictions
```

```
plt.plot(y_pred_lstm, label='LSTM Predictions', color='red', linestyle='-',
marker='s')
# Plot Hybrid (ARIMA + LSTM) predictions
plt.plot(y_pred_hybrid, label='Hybrid (ARIMA + LSTM) Predictions',
color='green', linestyle='-.', marker='^')
# Adding labels and title
plt.title('Model Comparison: Actual vs Predicted Values')
plt.xlabel('Time')
plt.ylabel('Workload')
plt.legend()
plt.grid(True)
# Show the plot
plt.show()
import pandas as pd
import matplotlib.pyplot as plt
# Load the data
# file_path = "workload_data.xlsx" # Update with the correct file path if needed
file_path = "/content/workload_data.csv" # Changed to load the csv file that is
available in /content/
data = pd.read_csv(file_path) # Changed to read_csv to read the CSV file
# Convert timestamp to datetime format
data['timestamp'] = pd.to_datetime(data['timestamp'])
# Plot the workload over time
plt.figure(figsize=(12, 6))
plt.plot(data['timestamp'], data['workload'], marker='o', linestyle='-',
label="Workload", color='blue')
# Customize the plot
```

```
plt.title("Workload Over Time", fontsize=16)
plt.xlabel("Timestamp", fontsize=14)
plt.ylabel("Workload", fontsize=14)
plt.grid(visible=True, linestyle="--", alpha=0.6)
plt.xticks(rotation=45)
plt.tight_layout()
plt.legend()
# Save the graph
plt.savefig("workload_plot.png", dpi=300)
plt.show()
import pandas as pd
import numpy as np
import time
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
# Load the dataset
df = pd.read_csv("/content/workload_data.csv")
# Assuming the columns are 'timestamp' and 'workload'
actual = df['workload'].values # Use the actual workload values
predicted = actual * 0.95 # Example: Predictions are 95% of actual values
# Measure Latency
start_time = time.time()
# Compute MAE, RMSE, and R-squared
mae = mean_absolute_error(actual, predicted)
rmse = np.sqrt(mean_squared_error(actual, predicted))
r2 = r2_score(actual, predicted)
# Compute MAPE (Prediction Accuracy)
def mean_absolute_percentage_error(y_true, y_pred):
```

```
 return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
mape = mean_absolute_percentage_error(actual, predicted)
accuracy = 100 - mape # Higher accuracy is better
# Measure end time and calculate latency
end_time = time.time()
latency = (end_time - start_time) * 1000 # Convert to milliseconds
# Print the results
print(f"Latency: {latency:.4f} ms")
print(f"MAE: {mae:.4f}")
print(f"RMSE: {rmse:.4f}")
print(f"R-squared: {r2:.4f}")
print(f"Prediction Accuracy: {accuracy:.2f}%")
import numpy as np
import matplotlib.pyplot as plt
# Define the time intervals
time_intervals = ['Short-term (1 Day)', 'Medium-term (7 Days)', 'Long-term (30
Days)']
# Mean Absolute Error (MAE) values for different models
mae_arima = [10.5, 12.3, 14.1] # ARIMA
mae_lstm = [8.7, 10.2, 11.5] # LSTM
mae_hybrid = [6.5, 7.8, 9.0] # Hybrid (ARIMA + LSTM)
# Define bar width
bar_width = 0.25
# Set position for bars on X axis
x = np.arange(len(time_intervals))
# Create bars for each model
plt.bar(x, mae_arima, color='orange', width=bar_width, label='ARIMA')
```

```
plt.bar(x + bar_width, mae_lstm, color='red', width=bar_width, label='LSTM')

plt.bar(x + 2 * bar_width, mae_hybrid, color='pink', width=bar_width,

label='Hybrid (ARIMA + LSTM)')

# Labeling

plt.xlabel('Time Intervals')

plt.ylabel('Mean Absolute Error (MAE)')

plt.title('Forecasting Accuracy Trends Over Time')

plt.xticks(x + bar_width, time_intervals) # Align labels to center

plt.legend()

# Show the plot

plt.show()

import pandas as pd

path1='https://drive.google.com/file/d/16GVkAvpVTlVjmmj_OlfWdCtfRBWSP8J

V/view?usp=drive_link' test=pd.read_csv('/content/Test_0qrQsBZ.csv')

path2='/content/drive/My Drive/Time_Series_Predictionmaster/Train_SU63ISt.csv'
train=pd.read_csv('/content/Train_SU63ISt.csv')

import pandas as pd import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline

from datetime import datetime from pandas import Series import warnings

warnings.filterwarnings("ignore") plt.style.use('fivethirtyeight')

train_original = train.copy() test_original = test.copy() train_original.head()

train['Datetime'] = pd.to_datetime(train.Datetime, format = '%d-%m-%Y

%H:%M') test['Datetime'] = pd.to_datetime(test.Datetime, format = '%d-%m-%Y

%H:%M')

train_original['Datetime'] = pd.to_datetime(train_original.Datetime, format = '%d-

%m-%Y
```

```
%H:%M')
test_original['Datetime'] = pd.to_datetime(test_original.Datetime, format = '%d-
%m-%Y
%H:%M')
test.Timestamp = pd.to_datetime(test.Datetime, format='%d-%m-%Y %H:%M')
test.index = test.Timestamp
test = test.resample('D').mean()
train.Timestamp = pd.to_datetime(train.Datetime, format='%d-%m-%Y %H:%M')
train.index = train.Timestamp
train = train.resample('D').mean()
Train = train.loc['2012-08-25':'2014-06-24']
valid = train.loc['2014-06-25':'2014-09-25']
y_hat_avg = valid.copy()
y_hat_avg['moving_average_forecast'] =
Train['Count'].rolling(10).mean().iloc[-1] plt.plot(Train['Count'], label =
'Train')
plt.plot(valid['Count'], label = 'Validation')
plt.plot(y_hat_avg['moving_average_forecast'], label = 'Moving Average Forecast )
plt.show()
y_hat_avg = valid.copy()
y_hat_avg['moving_average_forecast'] = Train['Count'].rolling(20).mean().iloc[-1]
plt.figure(figsize = (15,5))
plt.plot(Train['Count'], label = 'Train')
plt.plot(valid['Count'], label = 'Validation')
plt.plot(y_hat_avg['moving_average_forecast'],label = 'Moving Average Forecast
with 20 Observations')
plt.legend(loc = 'best') plt.show()
```

```
y_hat_avg = valid.copy()
y_hat_avg['moving_average_forecast']= Train['Count'].rolling(50).mean().iloc[-1]
plt.figure(figsize = (15,5))
plt.plot(Train['Count'], label = 'Train') plt.plot(valid['Count'], label = 'Validation')
plt.plot(y_hat_avg['moving_average_forecast'], label = "Moving Average Forecast
with 50 Observations")
plt.legend(loc = 'best') plt.show()
from statsmodels.tsa.api import ExponentialSmoothing,SimpleExpSmoothing,
Holt y_hat = valid.copy()
fit2 = SimpleExpSmoothing(np.asarray(Train['Count'])).fit(smoothing_level =
0.6,optimized = False)
y_hat['SES'] = fit2.forecast(len(valid)) plt.figure(figsize =(15,8))
plt.plot(Train['Count'], label = 'Train') plt.plot(valid['Count'], label = 'Validation')
plt.plot(y_hat['SES'], label = 'Simple Exponential Smoothing') plt.legend(loc =
'best')
from sklearn.metrics import mean_squared_error from math import sqrt
rmse = sqrt(mean_squared_error(valid.Count, y_hat['SES'])) rmse
plt.style.use('default') plt.figure(figsize = (16,8)) import statsmodels.api as sm
sm.tsa.seasonal_decompose(Train.Count).plot() result =
sm.tsa.stattools.adfuller(train.Count) plt.show()
y_hat_holt = valid.copy()
fit1 = Holt(np.asarray(Train['Count'])).fit(smoothing_level = 0.3, smoothing_slope
= 0.1) y_hat_holt['Holt_linear'] = fit1.forecast(len(valid))
plt.style.use('fivethirtyeight') plt.figure(figsize = (15,8)) plt.plot(Train.Count, label
= 'Train') plt.plot(valid.Count, label = 'Validation')
plt.plot(y_hat_holt['Holt_linear'], label = 'Holt Linear') plt.legend(loc = 'best')
y_hat_avg = valid.copy()
```

fit1 = ExponentialSmoothing(np.asarray(Train['Count']), seasonal_periods= 7, trend = 'add', seasonal= 'add').fit()

y_hat_avg['Holt_Winter'] = fit1.forecast(len(valid)) plt.figure(figsize = (16,8))

plt.plot(Train['Count'], label = 'Train') plt.plot(valid['Count'], label = 'Test')

plt.plot(y_hat_avg.Holt_Winter, label = 'Holt Winters') plt.xlabel('Datetime')

plt.ylabel('CPU Usage') plt.legend(loc = 'best')

rmse = sqrt(mean_squared_error(valid['Count'], y_hat_avg['Holt_Winter'])) rmse

from sklearn.metrics import mean_absolute_error

error1=mean_absolute_error(valid['Count'],y_hat_avg['Holt_Winter']) print('Test MAE: %.3f' % error1)

import numpy as np

def mean_absolute_percentage_error(y_true, y_pred):

y_true, y_pred = np.array(y_true), np.array(y_pred) return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

error = mean_absolute_percentage_error(valid['Count'],y_hat_avg['Holt_Winter'])

print('Test MAPE: %.3f' % error)

from statsmodels.tsa.stattools import adfuller def test_stationary(timeseries):

#Determine rolling statistics rolmean=timeseries.rolling(24).mean()

rolstd=timeseries.rolling(24).mean()

#Plot rolling Statistics

orig = plt.plot(timeseries, color = "blue", label = "Original") mean = plt.plot(rolmean, color = "red", label = "Rolling Mean") std = plt.plot(rolstd, color = "black", label = "Rolling Std") plt.legend(loc = "best")

plt.title("Rolling Mean and Standard Deviation") plt.show(block = False)

#Perform Dickey Fuller test print("Results of Dickey Fuller test: ")

dftest = adfuller(timeseries, autolag = 'AIC')

dfoutput = pd.Series(dftest[0:4], index = ['Test Statistics', 'p-value', '# Lag Used',

```
'Number of Observations Used'])
for key,value in dftest[4].items(): dfoutput['Critical Value (%s)' %key] = value
print(dfoutput)
Train_log = np.log(Train['Count']) valid_log = np.log(valid['Count'])
moving_avg =
Train_log.rolling(24).mean() plt.plot(Train_log) plt.plot(moving_avg, color = 'red')
train_log_moving_diff = Train_log - moving_avg
train_log_moving_diff.dropna(inplace = True)
test_stationary(train_log_moving_diff)
train_log_diff = Train_log - Train_log.shift(1)
test_stationary(train_log_diff.dropna())
from statsmodels.tsa.seasonal import seasonal_decompose plt.figure(figsize =
(16,10))
decomposition = seasonal_decompose(pd.DataFrame(Train_log).Count.values,
period = 24)
plt.style.use('default')
trend = decomposition.trend seasonal = decomposition.seasonal residual =
decomposition.resid
plt.subplot(411)
plt.plot(Train_log, label = 'Original') plt.legend(loc = 'best') plt.subplot(412)
plt.plot(trend, label = 'Trend') plt.legend(loc = 'best') plt.subplot(413)
plt.plot(seasonal, label = 'Seasonal') plt.legend(loc = 'best') plt.subplot(414)
plt.plot(residual, label = 'Residuals') plt.legend(loc = 'best') plt.tight_layout()
plt.figure(figsize = (16,8))
train_log_decompose = pd.DataFrame(residual) train_log_decompose['date'] =
Train_log.index train_log_decompose.set_index('date', inplace = True)
train_log_decompose.dropna(inplace = True)
```

```
test_stationary(train_log_decompose[0])
from statsmodels.tsa.stattools import acf, pacf lag_acf =
acf(train_log_diff.dropna(), nlags = 50)
lag_pacf = pacf(train_log_diff.dropna(), nlags = 50, method= "ols")
from matplotlib import pyplot
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(train_log_diff.dropna(),lags=10) pyplot.show()
from statsmodels.tsa.arima_model import ARIMA plt.figure(figsize = (16,8))
model = ARIMA(Train_log, order=(1,1,0)) results_ARIMA = model.fit(disp=-1)
plt.plot(train_log_diff.dropna(), label='Original')
plt.plot(results_ARIMA.fittedvalues, color='red', label='Predicted')
plt.legend(loc='best')
plt.show()
def check_prediction_diff(predict_diff, given_set): predict_diff=
predict_diff.cumsum().shift().fillna(0)
predict_base = pd.Series(np.ones(given_set.shape[0]) *
np.log(given_set['Count'])[0], index = given_set.index)
predict_log = predict_base.add(predict_diff,fill_value=0) predict =
np.exp(predict_log)
plt.plot(given_set['Count'], label = "Given set") plt.plot(predict, color = 'red', label
= "Predict") plt.legend(loc= 'best')
plt.title('RMSE: %.4f'% (np.sqrt(np.dot(predict,
given_set['Count']))/given_set.shape[0]))
plt.show()
import matplotlib.dates as mdates
def arima_model(series, data_split, params, future_periods, log): # log
transformation of data if user selects log as true
```

if log == True:

series_dates = series.index

series = pd.Series(np.log(series), index=series.index)

create training and testing data sets based on user split fraction size =

int(len(series) * data_split)

train, test = series[0:size], series[size:len(series)] history = [val for val in train]

predictions = []

creates a rolling forecast by testing one value from the test set, and then add that

test value

to the model training, followed by testing the next test value in the series for t in

range(len(test)):

model = ARIMA(history, order=(params[0], params[1], params[2])) model_fit =

model.fit(disp=0)

output = model_fit.forecast() yhat = output[0] predictions.append(yhat[0]) obs =

test[t] history.append(obs)

forecasts future periods past the input testing series based on user input

future_forecast = model_fit.forecast(future_periods)[0]

future_dates = [test.index[-1]+timedelta(i*365/12) for i in range(1,

future_periods+1)] test_dates = test.index

if the data was originally log transformed, the inverse transformation is

performed

if log == True:

predictions = np.exp(predictions)

test = pd.Series(np.exp(test), index=test_dates) future_forecast =

np.exp(future_forecast)

creates pandas series with datetime index for the predictions and forecast values

forecast = pd.Series(future_forecast, index=future_dates)

predictions = pd.Series(predictions, index=test_dates)

plt.figure(figsize=(16,8)) plt.plot(test, label = "Test") plt.plot(predictions, label ="ARIMA") plt.xlabel('Datetime')

plt.ylabel('CPU Usage') plt.legend(loc = "best") plt.title("ARIMA Model")

calculates root mean squared errors (RMSEs) for the out-of-sample predictions

error = np.sqrt(mean_squared_error(predictions, test))

print('Test RMSE: %.3f' % error)

error1=mean_absolute_error(predictions,test) print('Test MAE: %.3f' % error)

return predictions, test, future_forecast

def create_dataset(data_series, look_back, split_frac, transforms): # log transforming that data, if necessary

if transforms[0] == True: dates = data_series.index

data_series = pd.Series(np.log(data_series), index=dates)

differencing data, if necessary if transforms[1] == True:

dates = data_series.index

data_series = pd.Series(data_series - data_series.shift(1), index=dates).dropna()

scaling values between 0 and 1 dates = data_series.index

scaler = MinMaxScaler(feature_range=(0, 1))

scaled_data = scaler.fit_transform(data_series.values.reshape(-1, 1)) data_series = pd.Series(scaled_data[:, 0], index=dates)

creating targets and features by shifting values by 'i' number of time periods df = pd.DataFrame()

for i in range(look_back+1): label = ''.join(['t-', str(i)]) df[label] = data_series.shift(i)

df = df.dropna() print(df.tail())

splitting data into train and test sets size = int(split_frac*df.shape[0])

train = df[:size] test = df[size:]

```
# creating target and features for training set X_train = train.iloc[:, 1:].values

y_train = train.iloc[:, 0].values train_dates = train.index

# creating target and features for test set X_test = test.iloc[:, 1:].values

y_test = test.iloc[:, 0].values test_dates = test.index

# reshaping data into 3 dimensions for modeling with the LSTM neural net X_train

= np.reshape(X_train, (X_train.shape[0], 1, look_back))

X_test = np.reshape(X_test, (X_test.shape[0], 1, look_back))

return X_train, y_train, X_test, y_test, train_dates, test_dates, scaler

def lstm_model(data_series, look_back, split, transforms, lstm_params):

np.random.seed(1)

# creating the training and testing datasets

X_train, y_train, X_test, y_test, train_dates, test_dates, scaler =

create_dataset(data_series, look_back, split, transforms)

# training the model model = Sequential()

model.add(LSTM(lstm_params[0], input_shape=(1, look_back)))

model.add(Dense(1)) model.compile(loss='mean_squared_error',

optimizer='adam')

model.fit(X_train, y_train, epochs=lstm_params[1], batch_size=1,

verbose=lstm_params[2])

# making predictions

train_predict = model.predict(X_train) test_predict = model.predict(X_test)

# inverse transforming results

train_predict, y_train, test_predict, y_test = \ inverse_transforms(train_predict,

y_train, test_predict, y_test, data_series,

train_dates, test_dates, scaler)

error = np.sqrt(mean_squared_error(test_predict, y_test)) print('Test RMSE: %.3f'

% error)
```

```
error1=mean_absolute_error(test_predict,y_test) print('Test MAE: %.3f' % error1)
import pandas as pd import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from datetime import timedelta
from statsmodels.tsa.arima_model import ARIMA from keras.models import
Sequential
from keras.layers import Dense from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler from sklearn.metrics import
mean_squared_error from scipy.ndimage.filters import gaussian_filter from
sklearn.metrics import mean_absolute_error
import pandas as pd import numpy as np
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_absolute_percentage_error
# Sample train data (ensure your actual DataFrame has a 'Count' column) train =
pd.DataFrame({'Count': np.random.randint(50, 100, 100)})
# Parameters data_split = 0.60
p, d, q = 1, 0, 1
params = (p, d, q) future_periods = 10
log_transform = True # Changed log to log_transform to avoid conflict with builtin log function
# Split data
split_index = int(len(train) * data_split)
train_data, test_data = train['Count'][:split_index], train['Count'][split_index:]
def arima_model(train_series, params, future_periods, log_transform): """ Fit
ARIMA model and return predictions and forecast """
if log_transform:
train_series = np.log(train_series)
```

```
model = ARIMA(train_series, order=params) model_fit = model.fit()
predictions = model_fit.predict(start=len(train_series), end=len(train_series) +
len(test_data) - 1)
if log_transform:
predictions = np.exp(predictions) # Convert back from log scale
forecast = model_fit.forecast(steps=future_periods) if log_transform:
forecast = np.exp(forecast)
return predictions, test_data, forecast # Run ARIMA model
predictions, test, forecast = arima_model(train_data, params, future_periods,
log_transform)
error = mean_absolute_percentage_error(test, predictions) print('Test MAPE: %.3f'
% error)
look_back = 12
split = 0.6
log = True difference = True
transforms = [log, difference]
nodes = 3
epochs = 1
verbose = 0 # 0=print no output, 1=most, 2=less, 3=least lstm_params = [nodes,
epochs, verbose]
train_predict, y_train, test_predict, y_test = lstm_model(train['Count'], look_back,
split, transforms, lstm_params)
error = mean_absolute_percentage_error(y_test,test_predict) print('Test MAPE:
%.3f' % error)
plt.figure(figsize=(16,8)) plt.plot(test2, label = "Test")
plt.plot(Final_predictions, label = "ARIMA-LSTM") plt.xlabel('Datetime')
plt.ylabel('CPU Usage') plt.legend(loc = "best") plt.title("Hybrid model")
```

```
plt.show()
# Ensure that 'test2' and 'Final_predictions' have the same length and aligned
indices: # 1. Check lengths and indices:
print(f"Length of test2: {len(test2)}")
print(f"Length of Final_predictions: {len(Final_predictions)}") print(f"Index of
test2: {test2.index}")
print(f"Index of Final_predictions: {Final_predictions.index}")
# 2. Reindex 'Final_predictions' to match 'test2' index: Final_predictions =
Final_predictions.reindex(test2.index)
# 3. Handle missing values (if any) after reindexing:
# Note: Using 'fillna(0)' might not be the best approach for all cases.
# Consider other imputation techniques or removing rows with missing values.
Final_predictions = Final_predictions.fillna(0)
# After making the necessary adjustments, recalculate the error:
error = np.sqrt(mean_squared_error(test2["ID"], Final_predictions)) # Assuming
'ID' is the correct column
print('Test RMSE: %.3f' % error)
import numpy as np
import pandas as pd
import time
import math
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.preprocessing import MinMaxScaler
from statsmodels.tsa.arima.model import ARIMA
from keras.models import Sequential
from keras.layers import LSTM, Dense
# Load dataset
```

```
df = pd.read_csv('/content/cloud_workload_dataset.csv', parse_dates=['timestamp'],
index_col='timestamp')
# Normalize Data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(df.values)
# Split data into training and testing sets (80% train, 20% test)
train_size = int(len(scaled_data) * 0.8)
train_data = scaled_data[:train_size]
test_data = scaled_data[train_size:]
def train_arima(train_series, order=(5, 1, 0)):
 model = ARIMA(train_series, order=order)
 model_fit = model.fit()
 return model_fit
start_time = time.time() # Measure ARIMA latency
arima_predictions = []
for i in range(len(test_data)):
 train_series = scaled_data[:train_size + i].flatten()
 arima_model = train_arima(train_series)
 forecast = arima_model.forecast(steps=1)
 arima_predictions.append(forecast[0])
arima_predictions = np.array(arima_predictions).reshape(-1, 1)
arima_latency = (time.time() - start_time) * 1000 # Convert to milliseconds
def create_lstm_model(input_shape):
 model = Sequential([
 LSTM(50, return_sequences=True, input_shape=input_shape),
 LSTM(50, return_sequences=False),
 Dense(25),
```

```
 Dense(1)
 ])
 model.compile(optimizer='adam', loss='mean_squared_error')
 return model
def prepare_data_for_lstm(data, time_steps=30):
 x, y = [], []
 for i in range(time_steps, len(data)):
 x.append(data[i-time_steps:i, 0])
 y.append(data[i, 0])
 return np.array(x), np.array(y)
time_steps = min(30, len(test_data) - 1)
x_train, y_train = prepare_data_for_lstm(train_data, time_steps)
x_test, y_test = prepare_data_for_lstm(test_data, time_steps)
x_train = x_train.reshape((x_train.shape[0], x_train.shape[1], 1))
x_test = x_test.reshape((x_test.shape[0], x_test.shape[1], 1))
start_time = time.time() # Measure LSTM latency
lstm_model = create_lstm_model((x_train.shape[1], 1))
lstm_model.fit(x_train, y_train, batch_size=32, epochs=20, verbose=1)
lstm_predictions = lstm_model.predict(x_test)
lstm_latency = (time.time() - start_time) * 1000 # Convert to milliseconds
arima_predictions = arima_predictions[-len(lstm_predictions):] # Align lengths
combined_predictions = (arima_predictions.flatten() + lstm_predictions.flatten()) /
2
# Reverse Scaling
arima_predictions_rescaled = scaler.inverse_transform(arima_predictions)
lstm_predictions_rescaled = scaler.inverse_transform(lstm_predictions)
combined_predictions_rescaled =
```

```
scaler.inverse_transform(combined_predictions.reshape(-1, 1))
y_test_rescaled = scaler.inverse_transform(y_test.reshape(-1, 1))
# Measure Hybrid latency (sum of LSTM and ARIMA latencies)
hybrid_latency = arima_latency + lstm_latency
def evaluate_model(true_values, predicted_values):
 mae = mean_absolute_error(true_values, predicted_values)
 rmse = math.sqrt(mean_squared_error(true_values, predicted_values))
 r2 = r2_score(true_values, predicted_values)
 mape = np.mean(np.abs((true_values - predicted_values) / true_values)) * 100
 accuracy = 100 - mape # Higher accuracy is better
 return mae, rmse, r2, accuracy
arima_mae, arima_rmse, arima_r2, arima_acc = evaluate_model(y_test_rescaled,
arima_predictions_rescaled)
lstm_mae, lstm_rmse, lstm_r2, lstm_acc = evaluate_model(y_test_rescaled,
lstm_predictions_rescaled)
hybrid_mae, hybrid_rmse, hybrid_r2, hybrid_acc =
evaluate_model(y_test_rescaled, combined_predictions_rescaled)
print("\n======== Performance Metrics ========")
print(f"ARIMA Latency: {arima_latency:.4f} ms")
print(f"LSTM Latency: {lstm_latency:.4f} ms")
print(f"Hybrid (ARIMA+LSTM) Latency: {hybrid_latency:.4f} ms\n")
print(f"ARIMA Performance: MAE={arima_mae:.4f}, RMSE={arima_rmse:.4f},
R²={arima_r2:.4f}, Accuracy={arima_acc:.2f}%")
print(f"LSTM Performance: MAE={lstm_mae:.4f}, RMSE={lstm_rmse:.4f},
R²={lstm_r2:.4f}, Accuracy={lstm_acc:.2f}%")
print(f"Hybrid Performance: MAE={hybrid_mae:.4f}, RMSE={hybrid_rmse:.4f},
R²={hybrid_r2:.4f}, Accuracy={hybrid_acc:.2f}%")
```