# Multivariate AI Forecasting for Oracle Workload Using Transformers

Multivariate forecasting is **more advanced** than univariate forecasting because it considers multiple workload metrics **simultaneously**, learning how they interact over time. We’ll implement **Transformer-based multivariate AI forecasting** for **proactive database performance management**.

**Step 1: Expand Oracle Workload Data for Multivariate Analysis**

Instead of predicting **one metric at a time**, we now extract **multiple key workload metrics** as input variables.

**Extract Key Workload Metrics in SQL**

create\_table\_workload\_multivariate\_series.sql

CREATE TABLE workload\_multivariate\_series AS

SELECT collection\_time,

       EXTRACT(DAY FROM collection\_time) AS day\_of\_month,

       TO\_CHAR(collection\_time, 'D') AS day\_of\_week,

       EXTRACT(HOUR FROM collection\_time) AS hour\_of\_day,

       metric\_name,

       metric\_value

FROM system\_performance\_metrics

WHERE metric\_name IN ('CPU Usage (%)', 'Active Sessions', 'Buffer Cache Hit Ratio', 'Disk I/O Throughput', 'Redo Log Wait Time')

AND collection\_time >= SYSTIMESTAMP - INTERVAL '90' DAY

ORDER BY collection\_time;

This ensures:  
**Multiple workload metrics** (CPU, sessions, cache hit ratio, I/O, log waits).  
**Temporal information** (day, hour, weekday).  
**Chronologically ordered** dataset.

**Step 2: Train a Multivariate Transformer Model**

Instead of training **separate models** for each metric, we train **one Transformer model** that takes **multiple features** as input and predicts **multiple future workload values simultaneously**.

**Install Required Libraries**

pip install torch transformers datasets scikit-learn joblib cx\_Oracle pandas

**Python Code for Multivariate Transformer Training**

multivariate\_transformer\_training.py

import pandas as pd

import numpy as np

import cx\_Oracle

import torch

from torch import nn, optim

from transformers import TimeSeriesTransformerModel, TimeSeriesTransformerConfig

from sklearn.preprocessing import MinMaxScaler

import joblib

# Connect to Oracle & Load Data

dsn = cx\_Oracle.makedsn("your\_db\_host", "your\_db\_port", "your\_db\_service")

conn = cx\_Oracle.connect("your\_username", "your\_password", dsn)

query = """

SELECT collection\_time, metric\_name, metric\_value, day\_of\_week, hour\_of\_day

FROM workload\_multivariate\_series

"""

df = pd.read\_sql(query, conn)

conn.close()

# Pivot table: Convert rows → columns (multivariate format)

df = df.pivot(index="collection\_time", columns="metric\_name", values="metric\_value").reset\_index()

df.fillna(method="ffill", inplace=True)  # Fill missing values

# Normalize values for stability

scaler = MinMaxScaler()

scaled\_data = scaler.fit\_transform(df.iloc[:, 1:])  # Exclude time column

# Convert into sequences

sequence\_length = 24  # Use past 24 readings

X, y = [], []

for i in range(len(df) - sequence\_length):

    X.append(scaled\_data[i:i+sequence\_length, :])  # Use all workload metrics

    y.append(scaled\_data[i+sequence\_length, :])    # Predict all workload metrics

X, y = np.array(X), np.array(y)

# Convert to PyTorch tensors

X\_train, y\_train = torch.tensor(X[:-1000], dtype=torch.float32), torch.tensor(y[:-1000], dtype=torch.float32)

X\_test, y\_test = torch.tensor(X[-1000:], dtype=torch.float32), torch.tensor(y[-1000:], dtype=torch.float32)

# Transformer Model Configuration (Multivariate)

config = TimeSeriesTransformerConfig(

    d\_model=128,  # Larger hidden layer size

    n\_heads=8,   # More attention heads for complex patterns

    num\_encoder\_layers=4,

    num\_decoder\_layers=4,

    dropout=0.1,

    num\_features=X\_train.shape[-1]  # Number of workload metrics

)

model = TimeSeriesTransformerModel(config)

# Training

criterion = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(50):

    model.train()

    optimizer.zero\_grad()

    output = model(X\_train).squeeze()

    loss = criterion(output, y\_train)

    loss.backward()

    optimizer.step()

    print(f'Epoch {epoch+1}, Loss: {loss.item()}')

# Save Model & Scaler

torch.save(model.state\_dict(), "multivariate\_transformer\_workload.pth")

joblib.dump(scaler, "multivariate\_scaler.pkl")

This:  
Learns patterns across **multiple workload metrics**.  
Uses **24 past readings** for **multi-metric predictions**.  
Uses **multi-headed attention** for capturing interdependencies.  
Saves **multivariate Transformer model & scaler**.

**Step 3: Deploy Transformer API for Multivariate Predictions**

Now, we expose the trained Transformer model via a **Flask API** for real-time multivariate workload forecasting.

transformer\_model\_api.py

from flask import Flask, request, jsonify

import torch

import numpy as np

import joblib

from transformers import TimeSeriesTransformerModel, TimeSeriesTransformerConfig

app = Flask(\_\_name\_\_)

# Load Model & Scaler

config = TimeSeriesTransformerConfig(d\_model=128, n\_heads=8, num\_encoder\_layers=4, num\_decoder\_layers=4, dropout=0.1, num\_features=5)

model = TimeSeriesTransformerModel(config)

model.load\_state\_dict(torch.load("multivariate\_transformer\_workload.pth"))

model.eval()

scaler = joblib.load("multivariate\_scaler.pkl")

@app.route('/predict\_multivariate', methods=['POST'])

def predict():

    data = request.json['sequence']

    sequence = np.array(data).reshape(1, len(data), 5)  # 5 workload metrics

    sequence = torch.tensor(sequence, dtype=torch.float32)

    # Predict

    with torch.no\_grad():

        prediction = model(sequence).squeeze().numpy()

    # Convert back to original scale

    predicted\_values = scaler.inverse\_transform([prediction])

    return jsonify({

        'CPU Usage (%)': predicted\_values[0][0],

        'Active Sessions': predicted\_values[0][1],

        'Buffer Cache Hit Ratio': predicted\_values[0][2],

        'Disk I/O Throughput': predicted\_values[0][3],

        'Redo Log Wait Time': predicted\_values[0][4]

    })

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(host='0.0.0.0', port=5000)

Accepts **multiple workload metrics**.  
Uses **Transformers for multi-metric forecasting**.  
Returns **denormalized future workload values**.

**Step 4: Oracle PL/SQL Integration**

Now, we call the **Multivariate Transformer API from Oracle**.

multivariate\_transformer\_api\_call\_function.sql

CREATE OR REPLACE FUNCTION ml\_forecast\_multivariate (

    p\_metric\_names VARCHAR2

) RETURN CLOB IS

    v\_response CLOB;

    v\_sequence CLOB;

    v\_api\_url VARCHAR2(500) := 'http://your\_python\_server:5000/predict\_multivariate';

    v\_prediction CLOB;

BEGIN

    -- Fetch last 24 workload values for multiple metrics

    SELECT JSON\_ARRAYAGG(

        JSON\_OBJECT('CPU Usage (%)' VALUE cpu\_usage,

                    'Active Sessions' VALUE active\_sessions,

                    'Buffer Cache Hit Ratio' VALUE buffer\_cache,

                    'Disk I/O Throughput' VALUE disk\_io,

                    'Redo Log Wait Time' VALUE log\_wait\_time)

        ORDER BY collection\_time)

    INTO v\_sequence

    FROM (SELECT \* FROM workload\_multivariate\_series

          WHERE metric\_name IN ('CPU Usage (%)', 'Active Sessions', 'Buffer Cache Hit Ratio', 'Disk I/O Throughput', 'Redo Log Wait Time')

          ORDER BY collection\_time DESC FETCH FIRST 24 ROWS ONLY);

    -- Call Transformer API

    v\_response := http\_request(v\_api\_url, 'POST', '{"sequence": ' || v\_sequence || '}');

    -- Extract predicted workload values

    v\_prediction := JSON\_VALUE(v\_response, '$');

    RETURN v\_prediction;

END ml\_forecast\_multivariate;

/

Sends **past 24 readings for all workload metrics**.  
Gets **Transformer-based multivariate prediction**.  
Returns **forecasted values for proactive decision-making**.