# Implement **deep learning (DL) models** for workload forecasting

we'll replace traditional regression models with **Long Short-Term Memory (LSTM)** networks or **Transformers** to handle sequential time-series data more effectively.

**Step 1: Prepare Data for Deep Learning**

We will use LSTM (a type of recurrent neural network) to process sequential Oracle workload data. LSTMs excel in time-series forecasting, learning long-term dependencies better than traditional models.

**Modify Data Extraction for Time-Series Analysis**

We extract workload metrics **as a continuous sequence** rather than discrete values.

create\_table\_workload\_time.sql

CREATE TABLE workload\_time\_series AS

SELECT metric\_name, metric\_value, collection\_time

FROM system\_performance\_metrics

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

ORDER BY collection\_time;

This ensures **chronologically ordered** data for training.

**Step 2: Train an LSTM Model for Oracle Workload Prediction**

LSTMs require reshaped **3D input**:  
**(Samples, Time Steps, Features)**  
where:

* *Samples* = number of training examples
* *Time Steps* = how many past readings are used to predict the future
* *Features* = the number of workload parameters per reading

**Python Code for LSTM Model Training**

lstm\_model\_trainer.py

import pandas as pd

import numpy as np

import cx\_Oracle

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

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 metric\_value, collection\_time FROM workload\_time\_series"

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

conn.close()

# Convert timestamp to UNIX time

df['timestamp'] = pd.to\_datetime(df['collection\_time']).astype(int) / 10\*\*9

df = df.sort\_values(by='timestamp')

# Normalize values for stable training

scaler = MinMaxScaler()

df['scaled\_metric'] = scaler.fit\_transform(df[['metric\_value']])

# Convert into sequences

sequence\_length = 24  # Using last 24 readings (past hour if sampled every 2.5 minutes)

X, y = [], []

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

    X.append(df['scaled\_metric'].iloc[i:i+sequence\_length].values)

    y.append(df['scaled\_metric'].iloc[i+sequence\_length])

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

X = X.reshape(X.shape[0], X.shape[1], 1)  # Reshape for LSTM

# Train/Test Split

train\_size = int(len(X) \* 0.8)

X\_train, X\_test = X[:train\_size], X[train\_size:]

y\_train, y\_test = y[:train\_size], y[train\_size:]

# Build LSTM Model

model = Sequential([

    LSTM(64, return\_sequences=True, input\_shape=(X.shape[1], 1)),

    LSTM(32),

    Dense(16, activation='relu'),

    Dense(1)

])

model.compile(optimizer='adam', loss='mse')

model.fit(X\_train, y\_train, epochs=30, batch\_size=16, validation\_data=(X\_test, y\_test))

# Save Model & Scaler

model.save("lstm\_workload\_forecast.h5")

joblib.dump(scaler, "scaler.pkl")

This:  
Uses **LSTMs** to predict workload trends.  
**Normalizes** data for stability.  
Uses **past 24 readings** to predict the next value.  
Saves the trained model & scaler.

**Step 3: Deploy the LSTM Model with an API**

We create a Flask-based API so that Oracle can call predictions.

**Python API for Predictions**

api\_oracle\_call\_predictions.py

from flask import Flask, request, jsonify

import numpy as np

import joblib

import tensorflow as tf

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

# Load LSTM Model & Scaler

model = tf.keras.models.load\_model("lstm\_workload\_forecast.h5")

scaler = joblib.load("scaler.pkl")

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

def predict():

    data = request.json['sequence']

    # Convert data to numpy array & reshape for LSTM

    sequence = np.array(data).reshape(1, len(data), 1)

    # Predict

    prediction = model.predict(sequence)

    # Convert back to original scale

    predicted\_value = scaler.inverse\_transform(prediction)[0][0]

    return jsonify({'predicted\_value': predicted\_value})

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

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

Oracle sends a **sequence of workload values** (last 24 readings).  
LSTM **predicts the next workload value**.  
Returns **denormalized** (actual) workload prediction.

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

Now, Oracle needs to **call this API**.

oracle\_call\_api\_function.sql

CREATE OR REPLACE FUNCTION ml\_forecast\_lstm (

    p\_metric\_name VARCHAR2

) RETURN NUMBER IS

    v\_response CLOB;

    v\_sequence CLOB;

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

    v\_prediction NUMBER;

BEGIN

    -- Fetch last 24 workload values

    SELECT JSON\_ARRAYAGG(metric\_value ORDER BY collection\_time) INTO v\_sequence

    FROM (SELECT metric\_value FROM workload\_time\_series WHERE metric\_name = p\_metric\_name ORDER BY collection\_time DESC FETCH FIRST 24 ROWS ONLY);

    -- Call LSTM API

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

    -- Extract predicted value

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

    RETURN v\_prediction;

END ml\_forecast\_lstm;

/

Fetches the **last 24 workload values**.  
Sends them to **LSTM API for forecasting**.  
Returns **predicted workload** for Oracle analytics.

**Step 5: Automating AI Forecasts in Oracle**

We schedule **AI predictions every 30 minutes**.

ai\_prediction\_scheduler.sql

BEGIN

    DBMS\_SCHEDULER.create\_job (

        job\_name        => 'LSTM\_AI\_WORKLOAD\_JOB',

        job\_type        => 'PLSQL\_BLOCK',

        job\_action      => 'BEGIN generate\_lstm\_workload\_forecast; END;',

        start\_date      => SYSTIMESTAMP,

        repeat\_interval => 'FREQ=MINUTELY; INTERVAL=30',

        enabled         => TRUE

    );

END;

/

**Step 6: Enhancing the Chatbot with AI Forecasts**

The chatbot now **predicts workload issues** using LSTMs.

lstm\_forecast\_response\_procedure.sql

CREATE OR REPLACE PROCEDURE chatbot\_lstm\_forecast\_response(

    p\_user\_query IN VARCHAR2,

    p\_response OUT CLOB

) IS

    v\_predicted\_cpu NUMBER;

    v\_predicted\_sessions NUMBER;

    v\_alert\_msg VARCHAR2(500);

    v\_advice VARCHAR2(500);

BEGIN

    -- AI-powered forecasts

    v\_predicted\_cpu := ml\_forecast\_lstm('CPU Usage (%)');

    v\_predicted\_sessions := ml\_forecast\_lstm('Active Sessions');

    -- Generate alerts

    IF v\_predicted\_cpu > 80 THEN

        v\_alert\_msg := 'AI Forecast: High CPU usage predicted (' || v\_predicted\_cpu || '%).';

        v\_advice := 'Consider scaling resources or optimizing queries.';

    ELSIF v\_predicted\_sessions > 200 THEN

        v\_alert\_msg := 'AI Forecast: Surge in sessions expected (' || v\_predicted\_sessions || ').';

        v\_advice := 'Check application load balancing and tune queries.';

    ELSE

        v\_alert\_msg := 'No critical workload spikes predicted.';

        v\_advice := 'Monitor system performance.';

    END IF;

    -- Chatbot Response

    p\_response := v\_alert\_msg || ' Suggested action: ' || v\_advice;

END chatbot\_lstm\_forecast\_response;

/

**Predicts workload spikes 1 hour ahead.**  
**Chatbot proactively alerts DBAs.**  
**Suggests real-time solutions** based on AI predictions.