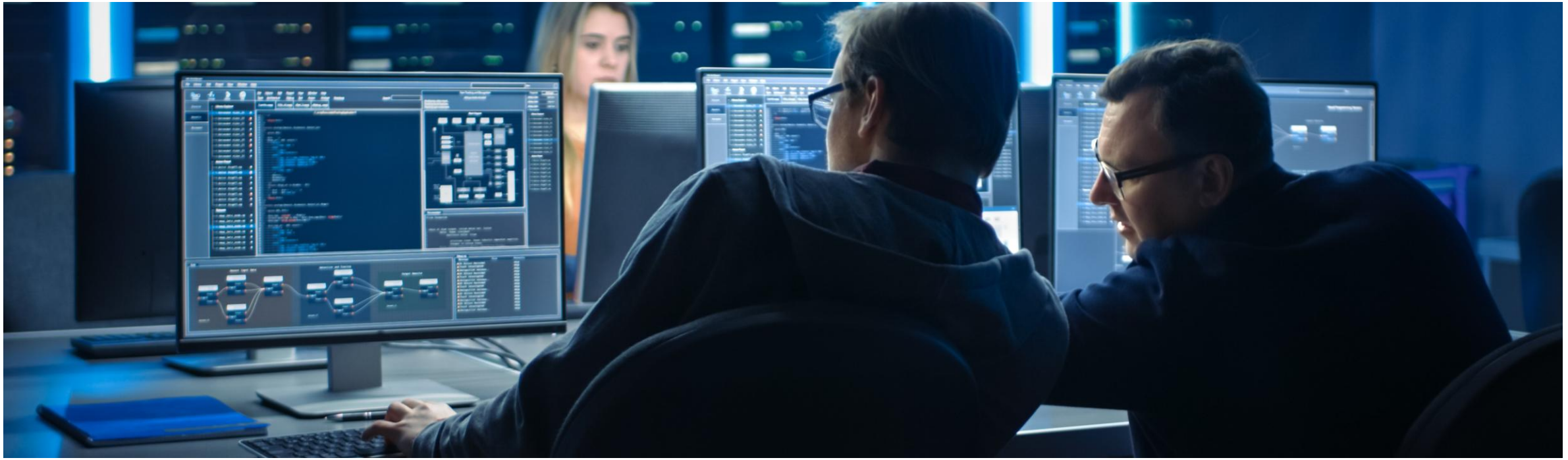
BOQ DATE DELIVERY PREDICTION (BOQ_DDP)
A RECURRENT NEURAL NETWORK FOR DELIVERY DATE PREDICTION

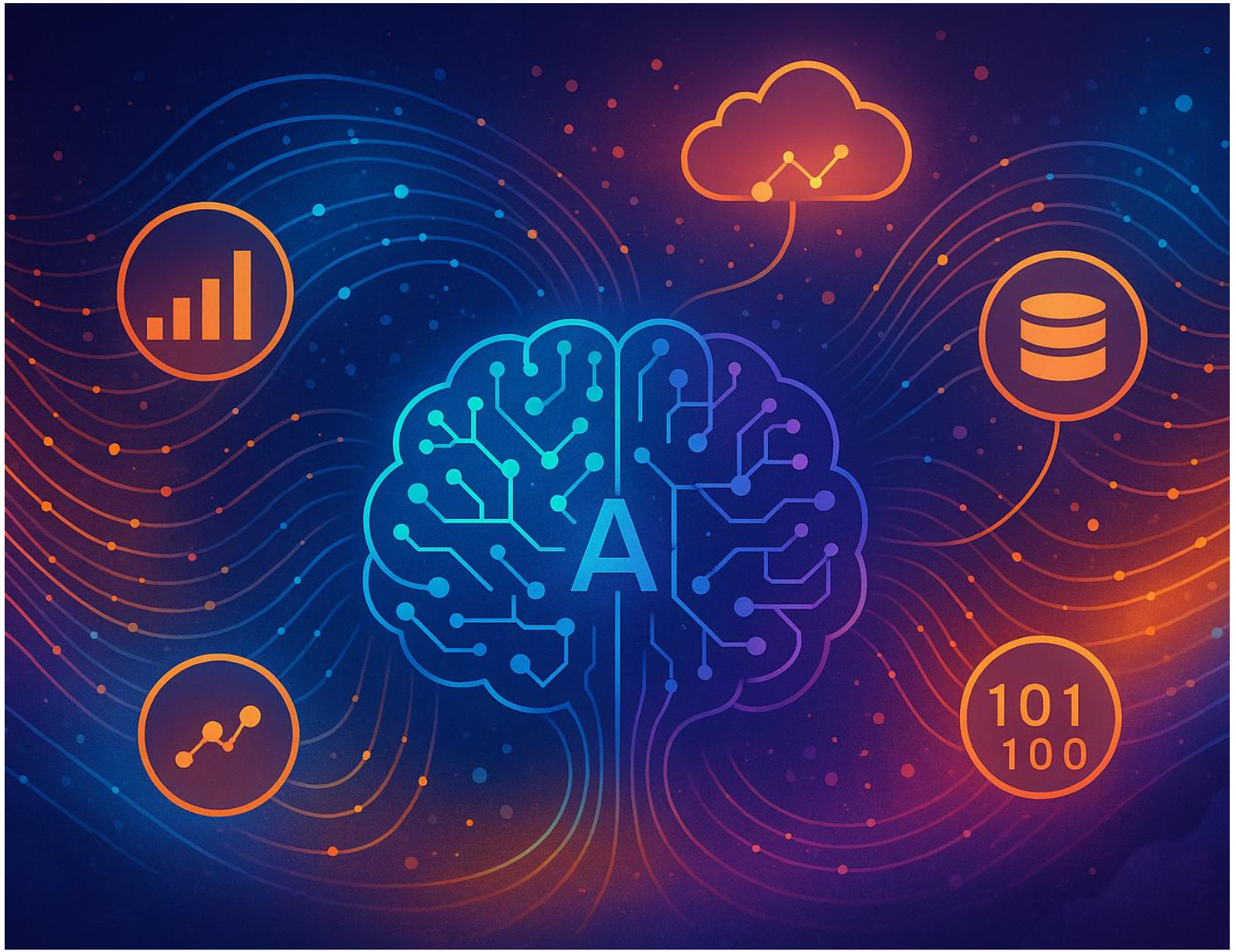


AGENDA

- Introduction
- Input data and preprocessing
- Architecture of RNN (LSTM)
- Training Process
- Results and their analysis
- Testing phase
- BoQ_DDP Web Application
- Conclusions

INTRODUCTION

1. Objective
    a. BoQ Delivery Date Prediction (BoQ_DDP) neural network, developed as part of a Proof-of-Concept (POC) to predict delivery dates for equipment items based on historical shipment data. Useful for Presales/PM/PEO
2. Dataset
    a. Source: Shipment delivery report 2024 and Q12025
    b. Size: 76,000 records
    c. Features: 14 variables
    - Useful for training
        - Customer Request Date
        - Order In Take Date
        - Actual Ship Date
        - LeadTimeDays
        - IntakeLag
3. Model Goals
    a. Build an accurate and generalizable model for managing time sequences of events
    b. Analyze feature importance
    c. Visualize training behavior and performance metrics
4. Development
    a. Framework: Colab of Google, Tensorflow
    b. Language: Python 3.13 and 3.10
    c. Libraries: NumPy, Pandas
    d. Collaborator: ChatGPT 4o Plus
5. Bibliography and Inspiration:
    a) *Python and Machine Learning A. Bellini, A. Guidi, McGraw-Hill.* Study case: *Predict the price of a stock, in our case Microsoft, starting from datasets available on Kaggle (https://www.kaggle.com/dgawlik/nyse)*



INPUT DATA AND PREPROCESSING

INPUT DATA AND PREPROCESSING

- The dataset used to train and validate the BoQ_DDP neural network is the report of the shipment of IP product (EUR zone) covering 2024 and Q1 2025 (csv format), containing:
  - Over 76,000 samples (instances)
  - 14 features for each item
  - Integer label (LeadTimeDays) as target
- Features include information on:
  - Item_ID
  - Customer Request Date
  - Order In Take Date
  - Actual Ship Date
  - LeadTimeDays (as Actual Ship Date - Customer Request Date)
  - IntakeLag (as Order In Take Date - Customer Request Date)

INPUT DATA AND PREPROCESSING

- These phases of preprocessing are done **OUTSIDE** of neural network training script.
  - **LeadTimeDays** (as Actual Ship Date - Customer Request Date)
    - If LeadTimeDays is strictly less than -15, then the predicted LeadTimeDays must be equal to 0 (in practice, the shipment arrives on time)
    - If LeadTimeDays takes a value between -15 (inclusive) and zero (inclusive), then the predicted LeadTimeDays must be equal to 5 (in practice, the shipment arrives after a week equal to 5 days)
    - If LeadTimeDays takes a value strictly greater than zero and less than or equal to 225, then the predicted LeadTimeDays is equal to the actual LeadTimeDays.
    - If LeadTimeDays takes a value strictly greater than 225, then the predicted LeadTimeDays is equal to 225.
  - **IntakeLag** (as Order In Take Date - Customer Request Date)
    - If IntakeLag is negative, its value will be 1 (in line with the customer's request).
    - If IntakeLag is positive, its value will be 2 (not in line with the customer's request).
    - If IntakeLag is zero, its value will be 0 (somewhat in line with the customer's request).

INPUT DATA AND PREPROCESSING

- These phases of preprocessing are done **INSIDE** of neural network training script
  - **Convert the Item_ID column** (which contains product identification strings) into a numeric representation that the neural network can use as input. This is crucial because neural models cannot directly handle text or categorical data in string format. The saved *item_id_encoder.pkl* file is essential during the deployment phase to ensure consistency. The same Item_ID must always be encoded in the same way, even when used in production.
    - Let's assume that Item_ID contains: ['3HE12345AA', '3HE67890BA', '3HE12345AA']. After the encoding:
    - ['3HE12345AA'] → 0
    - ['3HE67890BA'] → 1
    - The new column of Item_ID_enc is: [0, 1, 0]

INPUT DATA AND PREPROCESSING

- These phases of preprocessing are done **INSIDE** of neural network training script
  - **Sorting the dataset** by:
    - First for Item_ID_enc (i.e. groups all the same items together), then sort by Cust_Req_Ship_Date in ascending order within each group
    - Let's say you have the following unsorted dataset:
      - **Item_ID_enc Cust_Req_Ship_Date LeadTimeDays**
      - 2 2025-07-18 12
      - 1 2025-07-20 7
      - 2 2025-06-15 10
      - 1 2025-06-10 5
      - After the sorting:
      - **Item_ID_enc Cust_Req_Ship_Date LeadTimeDays**
      - 1 2025-06-10 5
      - 1 2025-07-20 7
      - 2 2025-06-15 10
      - 2 2025-07-18 12
    - This is essential for building the flowing timelines in the next steps, e.g. [t-2, t-1] → t.

INPUT DATA AND PREPROCESSING

- **Normalization**. when training a neural network (or a machine learning algorithm in general), the input data must be scaled and normalized. Main reason: different feature scales can slow down, prevent or damage learning. **MinMax** normalization scales the feature LeadTimeDays in the range [0, 1]. Summarizing: stable learning, balanced gradients, better generalization, all features treated equally. The saved *leadtime_scaler.pkl* file is used during prediction to correctly scale on input and do the inverse on output.

$$x_{\text{scaled}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \qquad x_{\text{original}} = x_{\text{scaled}} \times \left(x_{\text{max}} - x_{\text{min}}\right) + x_{\text{min}}$$

- Suppose LeadTimeDays is: [4, 5, 6, 10]. After MinMaxScaler (Min: 4, Max:10) → [0.0, 0.16, 0.33, 1.0] → Inverse Trasform: [4.0, 5.0, 6.0, 10.0].
- In practice, *leadtime_scaler.pkl* does not contain the data, but the mathematical rules (parameters) used to transform it.
- If you don't use the same scaler, the input data would be normalized differently, and the model would receive out-of-scale values → incorrect predictions.

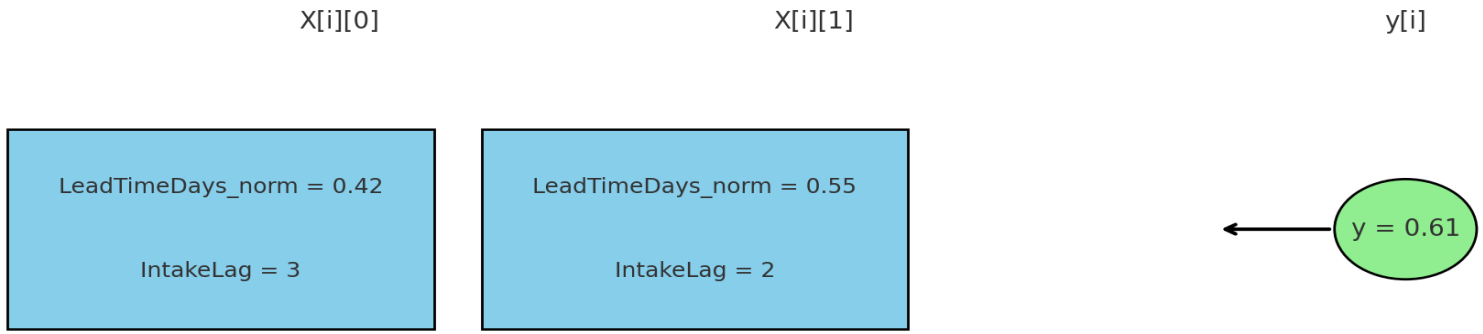
INPUT DATA AND PREPROCESSING

- Continuing
  - **Convert the DataFrame** into a time sequence dataset for each Item_ID, useful for training sequential models such as LSTM or GRU (sliding windows)
    - Suppose that for a certain Item_ID there are the following values:
      - LeadTimeDays_norm: [0.4, 0.5, 0.6, 0.8]
      - IntakeLag: [3, 2, 4, 1]
    - The complete sequence of features will be:
      - [[0.4, 3], [0.5, 2], [0.6, 4], [0.8, 1]]
    - Then with n_steps = 2 we create:
      - X[0] = [[0.4, 3], [0.5, 2]] → y[0] = 0.6
      - X[1] = [[0.5, 2], [0.6, 4]] → y[1] = 0.8

X is a 3D array: (number of sequences, n_steps, number of features)
y is a 1D array: target corresponding to each sequence
n_steps = 2 means the model will use 2 previous rows to make a prediction

INPUT DATA AND PREPROCESSING

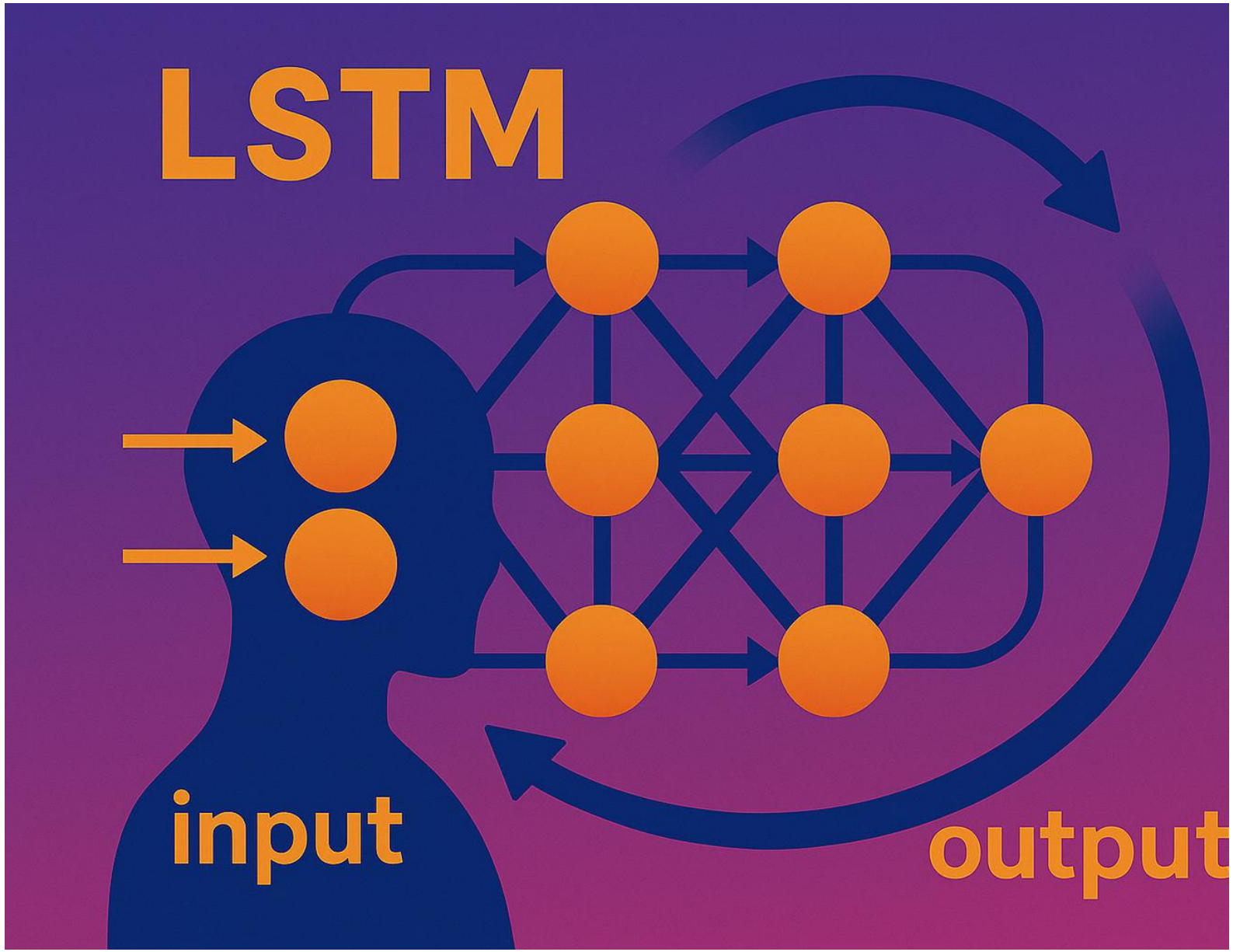
- This representation clearly shows how the data is packaged for training the LSTM/GRU model (see later).



- Here's a representation with a real-world numerical example:
  - The sequence X[i] is composed of two time steps:
    - X[i][0]: LeadTimeDays_norm = 0.42, IntakeLag = 3
    - X[i][1]: LeadTimeDays_norm = 0.55, IntakeLag = 2
  - The target value y[i] is 0.61, i.e., the value of LeadTimeDays_norm at the next time step.

INPUT DATA AND PREPROCESSING

- Continuing
  - **Splits the dataset** into a training set (70%) and a validation set (30%), based on a specified percentage. Training set is used for training phase; validation set is used for validation phase. Validation is used to verify that the training carried out with the training set does not generate overfitting.



ARCHITECTURE OF RNN (GRU+LSTM)

ARCHITECTURE OF RNN (GRU+LSTM)

- Why GRU+LSTM for Lead Time Prediction?
  a. The problem is sequential/temporal in nature: lead time prediction relies on past chronological data per item (e.g., IntakeLag, historical LeadTimeDays).
  b. Recurrent Neural Networks (RNNs) are suitable for sequences but suffer from long-term memory issues.
  c. GRU and LSTM architectures enhance memory handling and temporal dependency learning.
  d. The GRU + LSTM combination leverages:
     1. The **speed and simplicity** of GRU for short sequences.
     2. The **expressive power** of LSTM for learning deeper temporal patterns.
  e. In this project, GRU+LSTM was chosen to:
     1. Handle lead time variability.
     2. Balance model complexity and predictive power.

ARCHITECTURE OF RNN (GRU+LSTM)

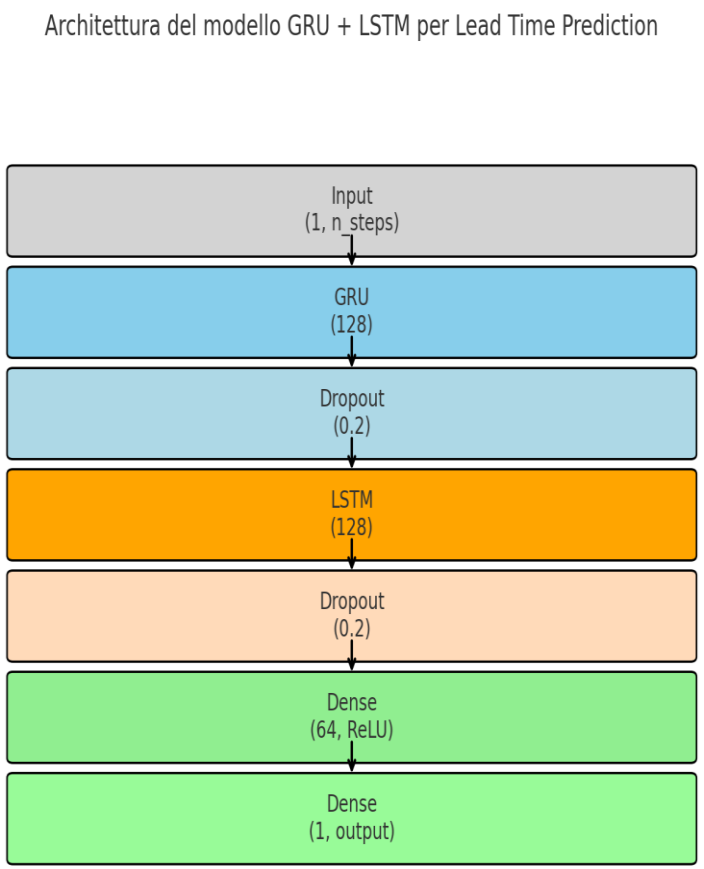
- How Does GRU+LSTM Work (Conceptually)?
  a. **GRU (Gated Recurrent Unit)**:
     1. Uses two gates: update and reset.
     2. Fewer parameters than LSTM; faster to train.
  b. **LSTM (Long Short-Term Memory)**:
     1. Uses three gates: input, forget, and output.
     2. Better suited for long-range temporal dependencies.
  c. **In our model**:
     1. GRU processes the raw input and returns a sequence of representations.
     2. LSTM receives the sequence and learns a compact representation.
     3. Dense layers produce the final prediction.

ARCHITECTURE OF RNN (GRU+LSTM)

- Key Characteristics of Our GRU+LSTM Model
  - **Sequential Architecture**:
    - GRU (128 units)
    - Dropout (0.2)
    - LSTM (128 units)
    - Dropout (0.2)
    - Dense (64, ReLU activation)
    - Dense (1, output)
  - **Design Rationale**:
    - GRU reduces early-stage overfitting.
    - LSTM captures deeper temporal sequences.
    - Dropout regularizes training.
    - Dense layer adds non-linearity and supports regression.

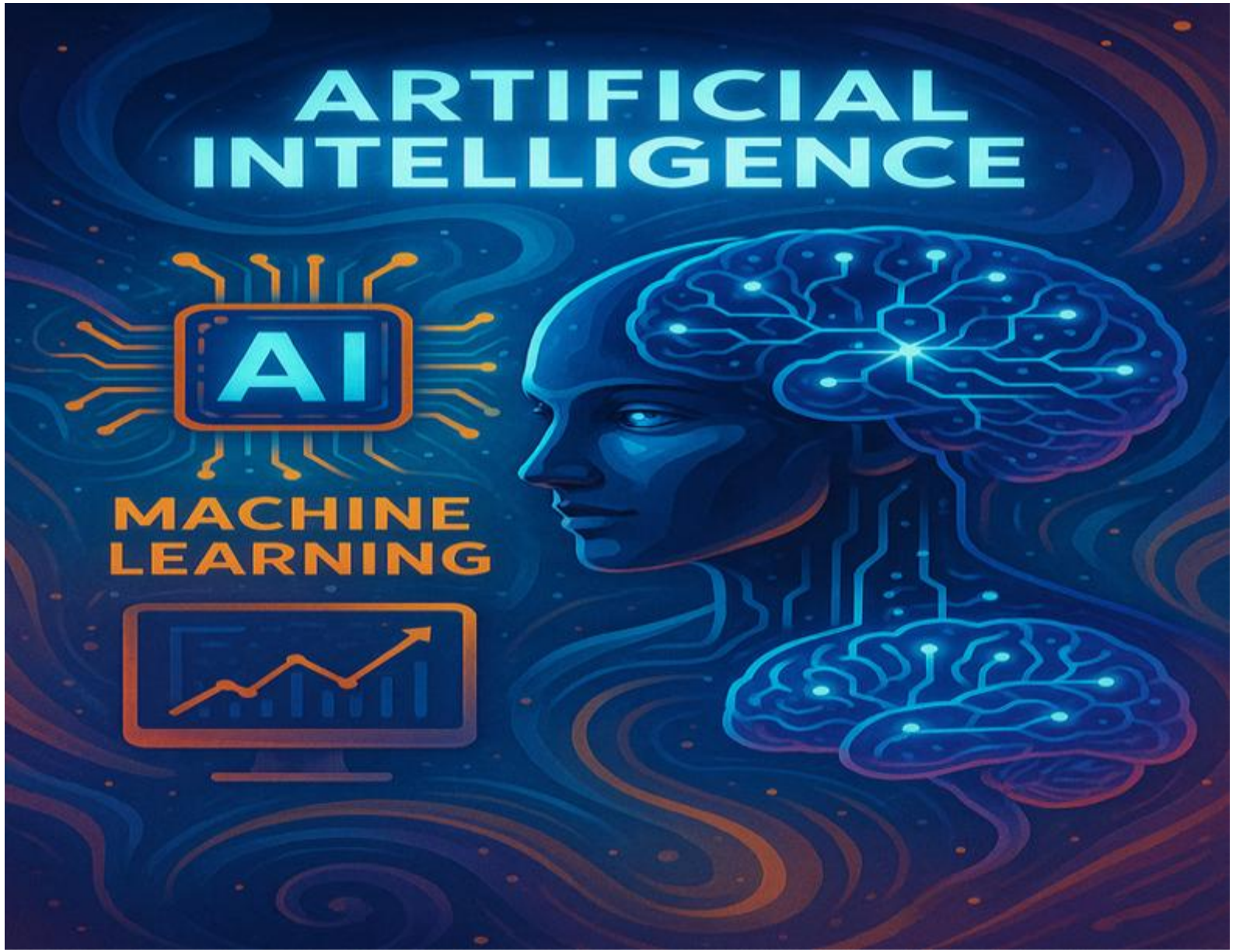
ARCHITECTURE OF RNN (GRU+LSTM)

- Simplified Graphical Representation of the Model
  - Block Diagram:
    1. Input: time sequences (LeadTimeDays_norm, IntakeLag)
    2. GRU Layer: 128 units → intermediate representation
    3. Dropout
    4. LSTM Layer: 128 units → deep temporal learning
    5. Dropout
    6. Dense (64, ReLU)
    7. Final Output: Dense (1)



ARCHITECTURE OF RNN (GRU+LSTM)

- Conclusion
  - The GRU+LSTM combination is a balanced solution for time-series prediction problems.
  - It allows:
    - Noise reduction in early-stage data (GRU)
    - Retention of useful sequence memory (LSTM)
    - Inference from historical component-based patterns.
  - Results indicate a good trade-off between accuracy and computational complexity.



TRAINING PROCESS, RESULTS AND ANALYSIS

TRAINING PROCESS, RESULTS AND ANALYSIS

- Model Compilation
  - Model compiled with the following configuration:
    - Loss function: Mean Squared Error (MSE), suitable for a regression problem

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

    - Optimizer: Adam (learning rate = 0.001), with an initial learning rate of 0.001. A balanced value between learning speed and stability
    - Metric: Mean Squared Error, again MSE is used for monitoring during training/validation
- EarlyStopping Callback
  - To prevent overfitting and unnecessary training:
    - Monitor: *val_mean_squared_error,* check the mean squared error on the validation data
    - Patience: 5 epochs, if there is no improvement for 5 consecutive epochs, training stops
    - Automatically restores best weights, upon completion, it automatically restores the epoch weights with the best *val_mse value*

TRAINING PROCESS, RESULTS AND ANALYSIS

- Training Phase
  - Model is trained using:
    - Training set: X_train, y_train, training data
    - Validation set: X_test, y_test, Validation performed at each epoch to monitor *val_mse*
    - Epochs: 50, Maximum number of epochs (complete cycles on the dataset)
    - Batch size: 32, the model updates the weights every 32 examples
    - Callback: EarlyStopping, activates the early stopping logic
- Training History
  - Output of *model.fit()* is stored in history:
    - Tracks epoch-wise MSE and val_MSE
    - Useful for plotting learning curves
    - Helps evaluate training quality

TRAINING PROCESS, RESULTS AND ANALYSIS

- Visual Flowchart

TRAINING PROCESS, RESULTS AND ANALYSIS

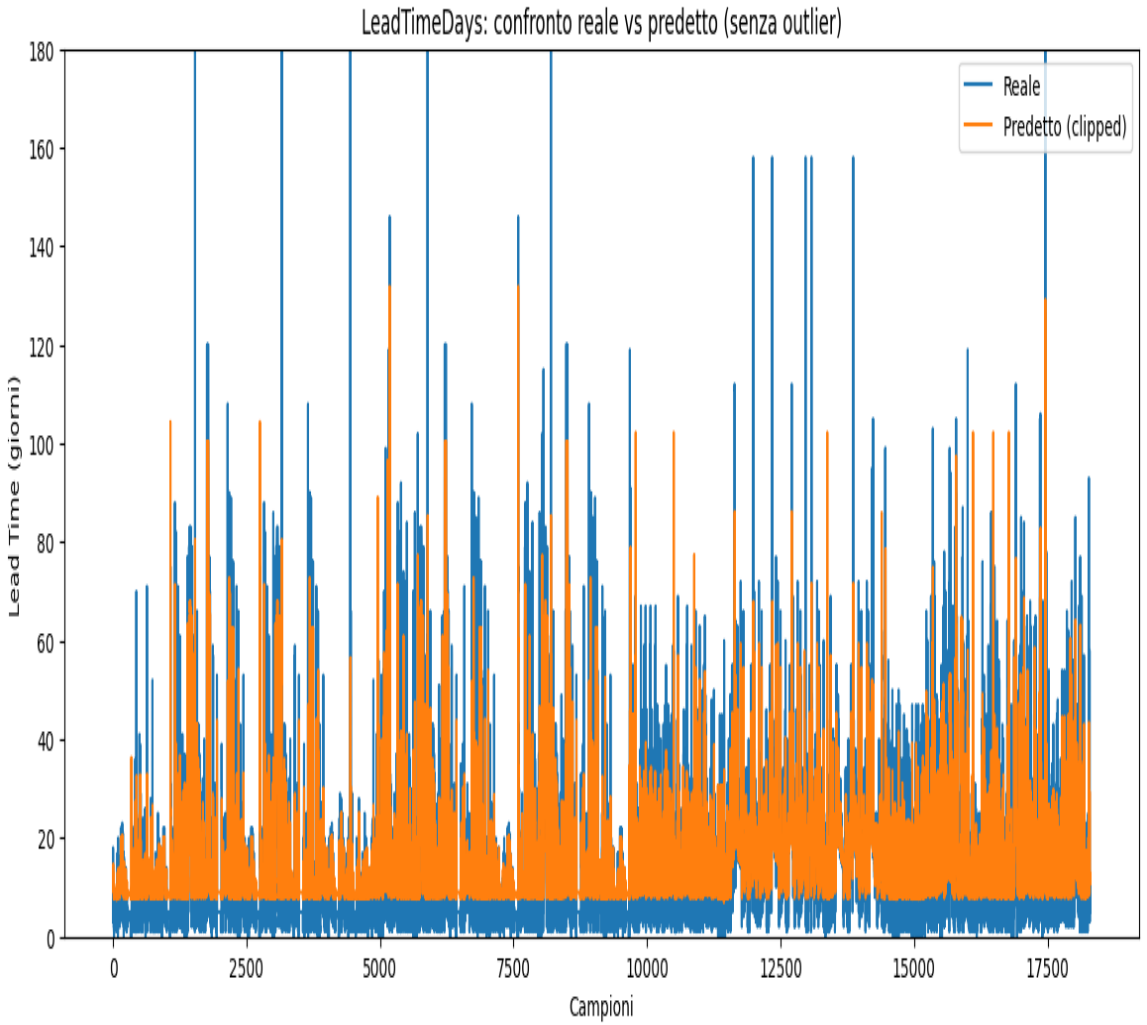
- Final Model Performance Metrics
  - MSE: 175.65 (days²) → Mean Squared Error
  - RMSE: 13.25 days → Root Mean Squared Error
  - MAE: **8.12 days** → Mean Absolute Error

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

  - MAE (Mean Absolute Error)
    - *Answers to question:* "On average, by how many days does the model miss the prediction?"
  - RMSE (Root Mean Squared Error)
    - *Answers to question* : "How much does the model miss on average, giving more weight to large errors?"
  - MSE (Mean Squared Error)
    - *Answers to question* : "What is the average variance of the errors (in days²) made by the model?" not immediate, loss function

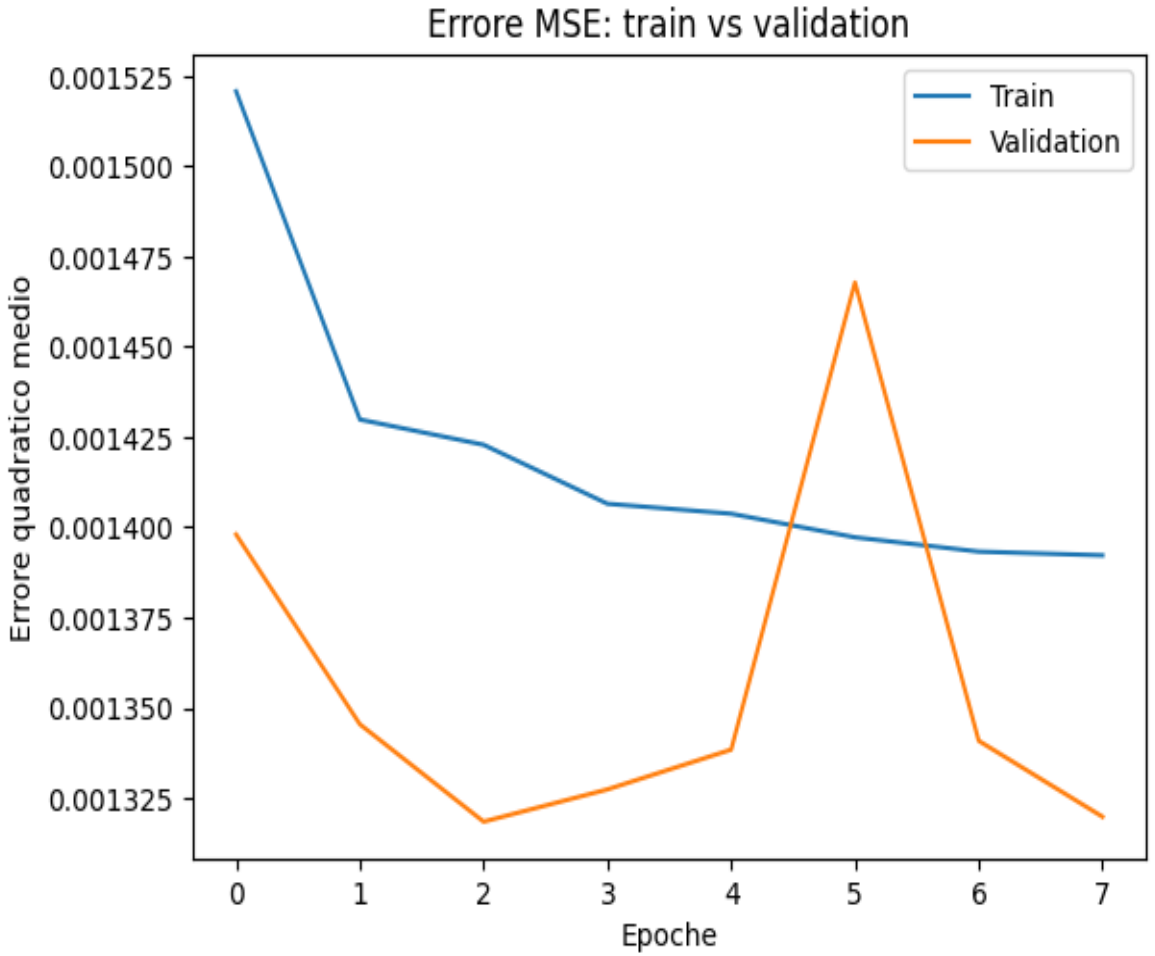
TRAINING PROCESS, RESULTS AND ANALYSIS

- LeadTimeDays: Actual vs Predicted
  - The model **closely follows the overall trend** of actual lead times
  - Predicted values are **clipped** to prevent unrealistic outliers
  - Slight underestimation in high-peak cases, but performance is solid overall
- Analysis: Actual vs Predicted
  - Strong accuracy for **low and medium** lead times
  - Slight underestimation in **high peaks**, handled with clipping
  - Best predictions occur in **densely represented regions** of the dataset



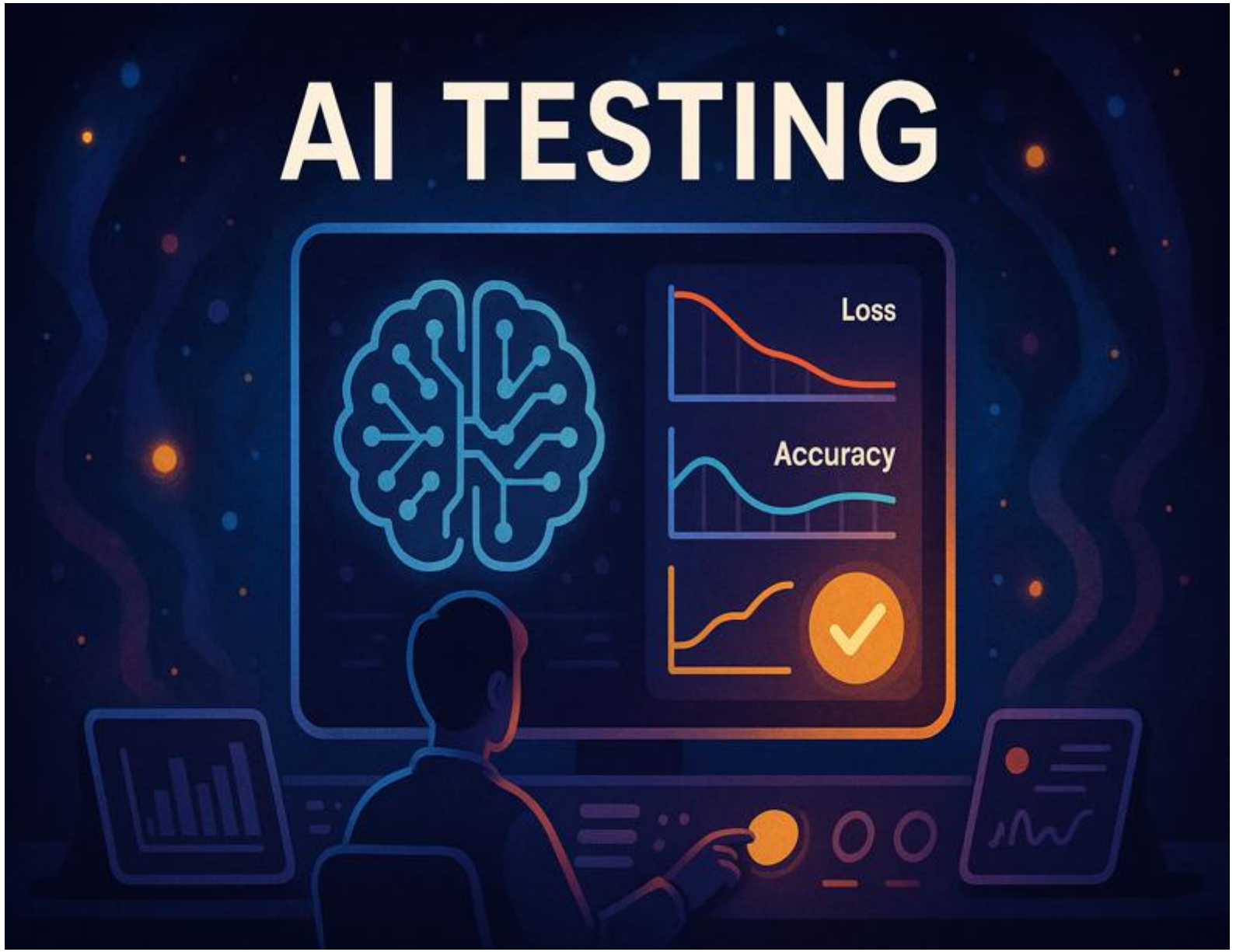
TRAINING PROCESS, RESULTS AND ANALYSIS

- Mean Squared Error: Train vs Validation
  - MSE steadily decreases for both training and validation sets
  - No signs of overfitting
  - EarlyStopping triggered after 7 epochs with best weights restored



TRAINING PROCESS, RESULTS AND ANALYSIS

- Training Conclusions
  - Training process is stable and well-regularized
  - EarlyStopping stopped at the optimal moment (epoch 7)
  - The model has robust predictive capacity, with MAE < 10 days
  - Ready for production usage



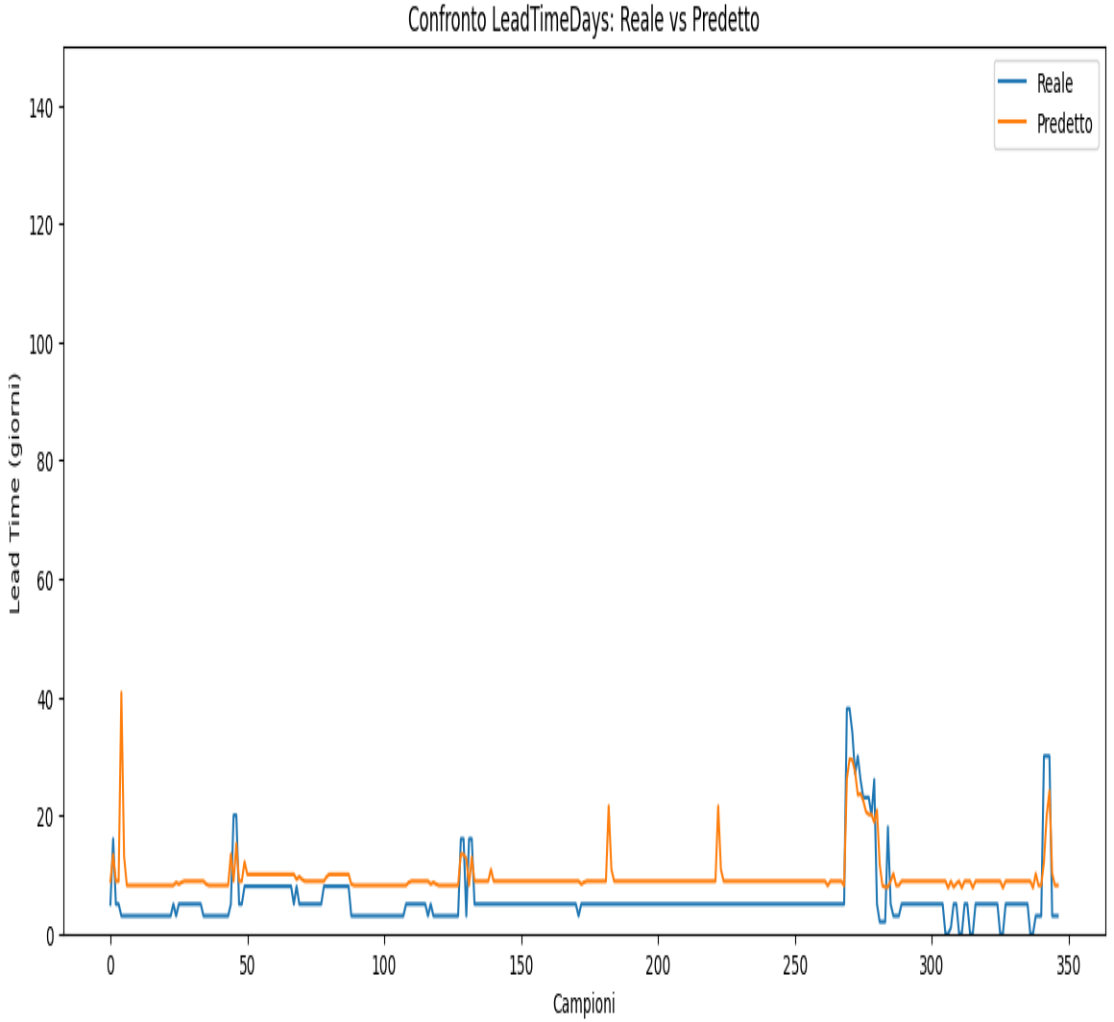
TESTING PHASE

TESTING PHASE

- Script Workflow Overview
- The test script performs the following sequential steps:
  - Load LSTM-GRU trained model (.keras)
  - Upload test dataset with delivery history (about 470 items)
  - Encode categorical variable Item_ID
  - Normalize LeadTimeDays using saved scaler
  - Filter items with at least 3 records
  - Generate time sequences with IntakeLag + LeadTimeDays_norm
  - Predict normalized LeadTimeDays using the model
  - Reverse normalization to obtain results in days
  - Plot predictions vs actuals and calculate error metrics

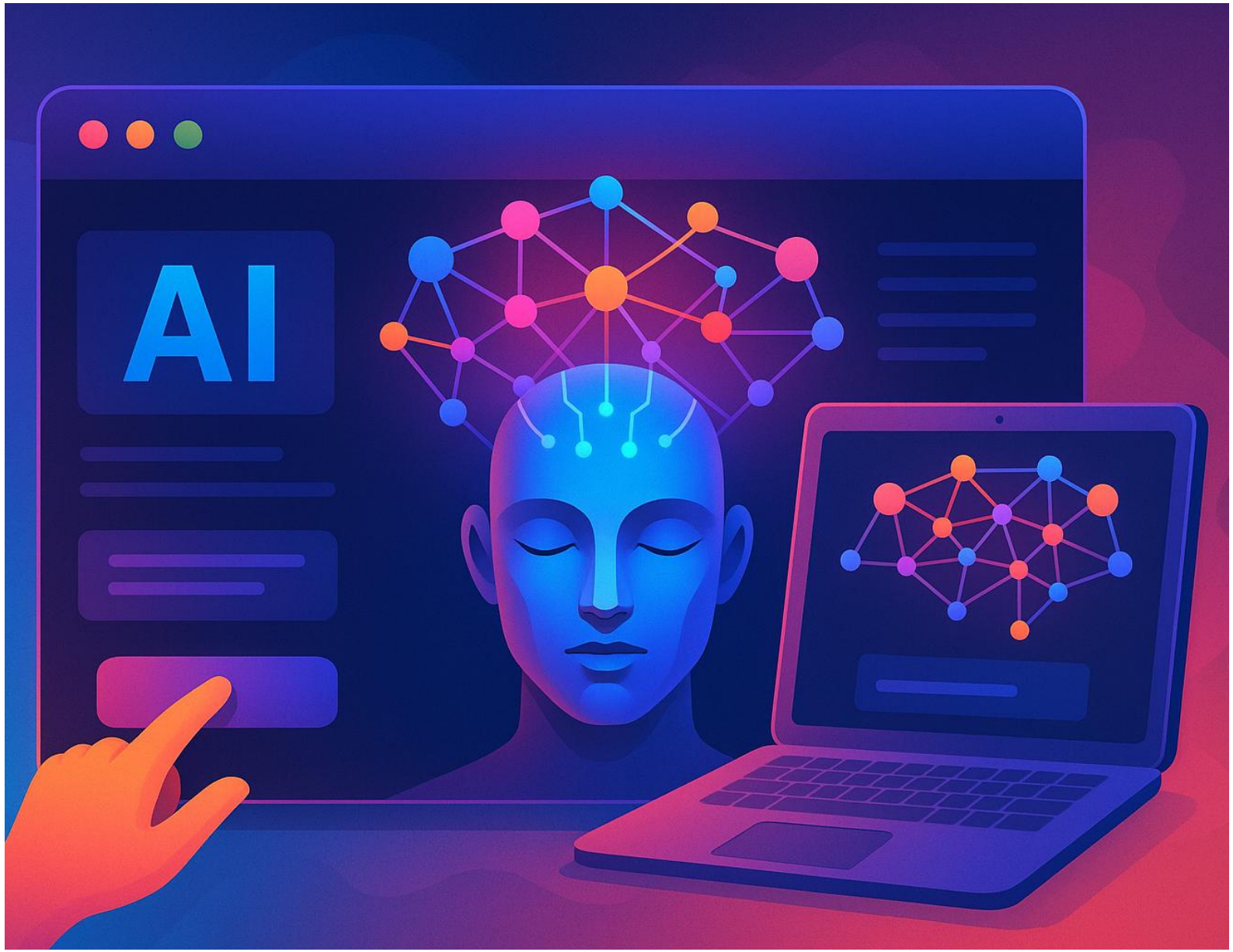
TESTING PHASE

- Predicted vs Actual Lead Time
- Graphical comparison of predicted vs actual delivery lead time (in days):
  - 🔵 Blue line: Ground truth from dataset
  - 🟠 Orange line: Model predictions
  - 📌 Observations:
    - Model is smooth, avoids noise
    - Accurate in normal ranges (5–15 days)
    - Underestimates delivery delays (spikes >30d)



TESTING PHASE

- Test Metrics Summary
- Final performance metrics on the test dataset:
  - MAE (Mean Absolute Error): 6.51 days → Average error, robust to outliers. Good average error tolerance (**MAE < 7d**)
  - RMSE (Root Mean Squared Error): 7.72 days → Penalizes large deviations
  - MSE (Mean Squared Error): 59.66 $days^2$→ Measures overall error dispersion

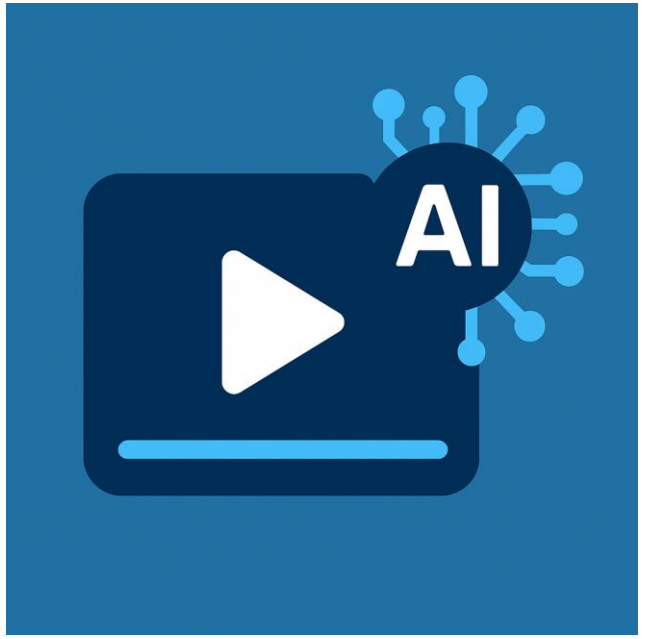


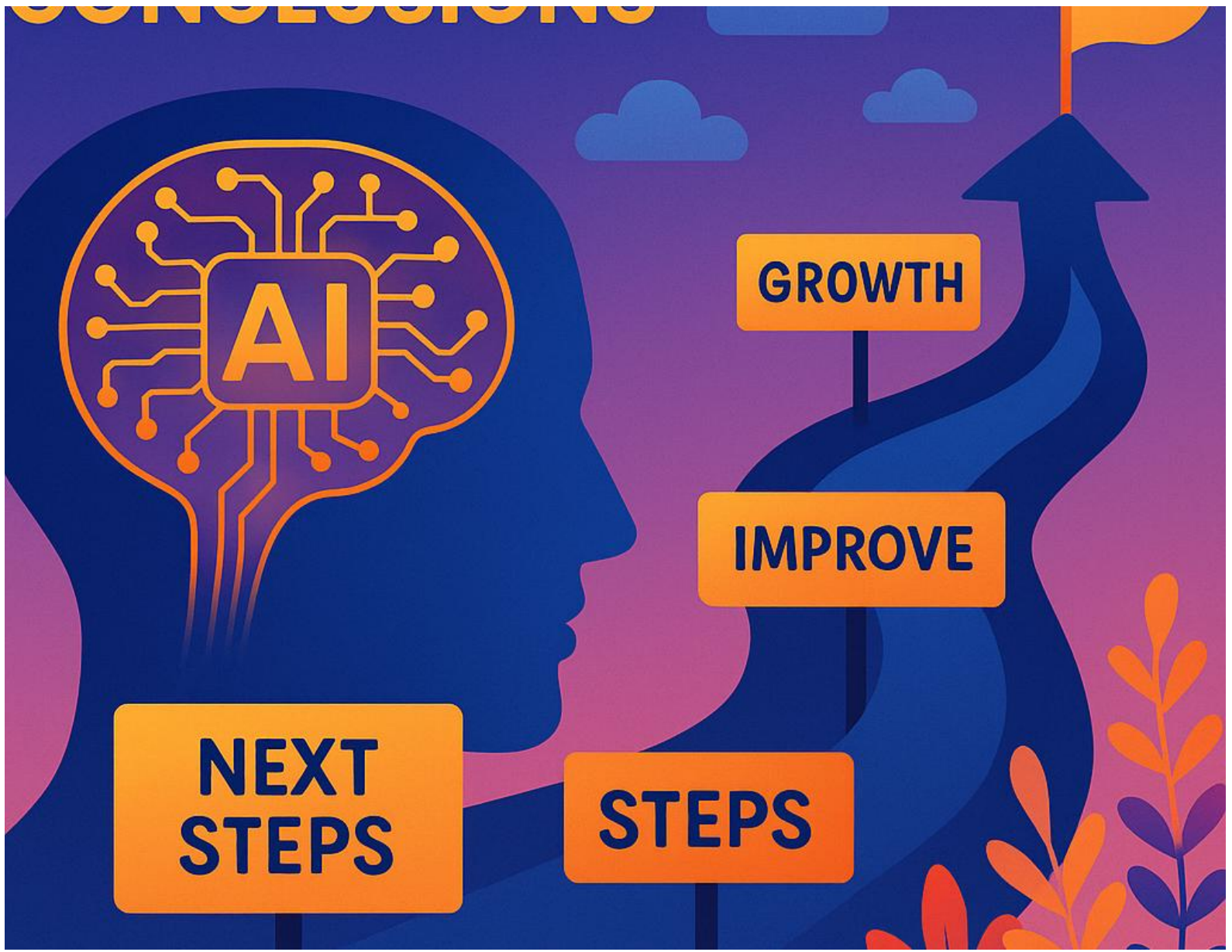
BOQ DATE DELIVERY PREDICTION WEB APPLICATION

BOQ DATE DELIVERY PREDICTION WEB APPLICATION

- A web application developed with Streamlit for predicting the delivery date of a BoQ using a pre-trained LSTM-GRU neural network.
- Key functionalities:
  - Upload of an Excel file with Item_ID entries to be evaluated
  - Input of two dates: Customer Request Date (CRD) and Order Intake Date (OID)
  - Calculation of IntakeLag and sequence generation for each item
  - Prediction of Lead Time (in days) using the neural model
  - Automatic calculation of the predicted delivery date (excluding weekends)
  - Tabular display of predicted dates with confidence level
  - Informative messages about model performance and options to reset or exit the app

Demo





CONCLUSION

CONCLUSION

- Benefit
  - Captures time-based patterns with LSTM-GRU
  - Good accuracy: MAE ≈ 8.12 days, RMSE ≈ 13.25 days
  - Works well with just 2 input features
  - Generalizes across many Item_IDs
- Future Improvements
  - Add more features (e.g., category, forecast type)
  - Handle extreme delays more accurately
  - Cluster similar items for specialized models
  - Set up periodic retraining with new data

THANK YOU

- Corrado Vaccaro
- Senior Customer Program Manager, PMP®
- corrado.vaccaro@nokia.com