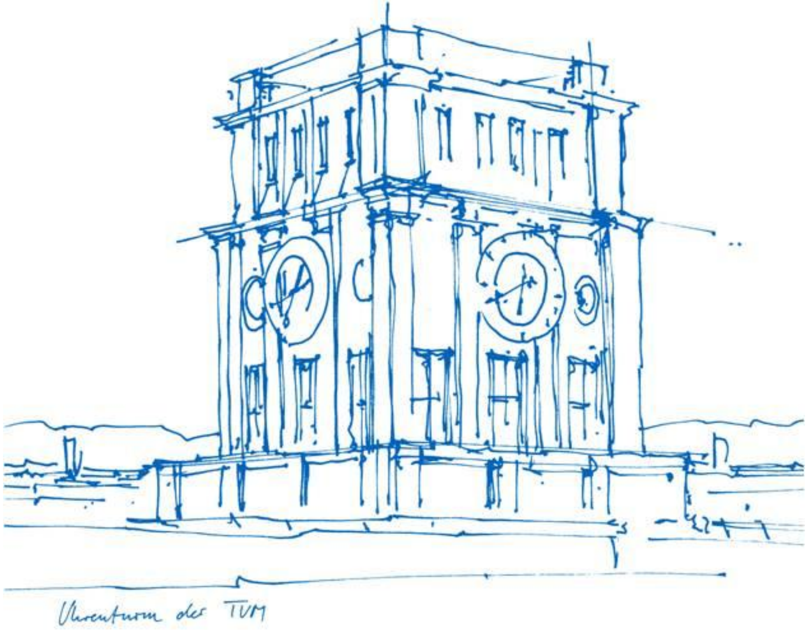
Predicting IT Incident Resolution Time

Team: Team 5

Members:

- Hayrettin İlbey GÜNGÖR — 03812227
- Hamza KALE — 03806213
- Anil Emre MENTES — 03786601



Content

Motivation & Problem Statement

1

Motivation

Why Predicting Incident Resolution Time Matters

- 🕒 **Operational Efficiency**
  Understanding expected resolution time helps optimize scheduling and resource allocation in support teams.

- 📈 **Performance Monitoring**
  Anticipated resolution times can be used to assess SLA compliance and team productivity.

- 💡 **Data-Driven Decision Making**
  Enables IT managers to act proactively on bottlenecks and workload distribution.

- 😌 **User Satisfaction**
  Accurate predictions lead to better communication with stakeholders and improved end-user trust.

Problem Statement & Goals

Problem Statement:
In IT service management, delays in resolving incidents can lead to service disruption and customer dissatisfaction.
Our goal is to **predict the resolution time (in hours)** for an incident based on historical event data in any stage.

Project Goals:

- Understand incident log data and its resolution time behavior.
- Apply exploratory and statistical analysis to uncover influencing factors.
- Build multiple machine learning models to predict resolution time.
- Evaluate and compare model performances using standard metrics.
- Identify the best approach for deployment in real-time IT support systems.

Domain & Data Overview

2

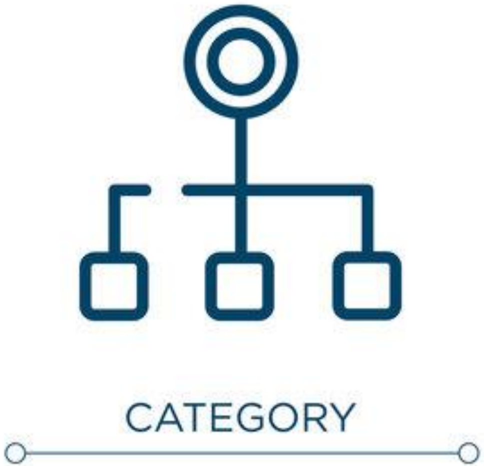
Domain & Data Overview

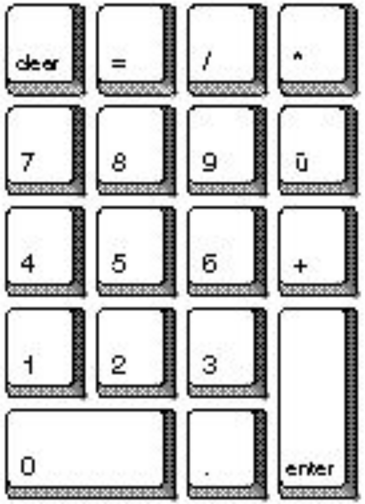
Domain: ITIL-based incident management; resolution time critical for SLA compliance

Dataset: 141,712 records × 36 features, each row = one incident state/update.

```
Dataset Shape: (141712, 36)

Data Types:
 number                      object
incident_state               object
active                         bool
reassignment_count            int64
reopen_count                  int64
sys_mod_count                 int64
made_sla                       bool
caller_id                    object
opened_by                    object
opened_at                    object
sys_created_by               object
sys_created_at               object
sys_updated_by               object
sys_updated_at               object
contact_type                 object
location                     object
category                     object
subcategory                  object
u_symptom                    object
cmdb_ci                      object
impact                       object
urgency                      object
priority                     object
assignment_group             object
assigned_to                  object
knowledge                      bool
u_priority_confirmation        bool
notify                       object
problem_id                   object
rfc                          object
vendor                       object
caused_by                    object
closed_code                  object
resolved_by                  object
resolved_at                  object
closed_at                    object
```

Domain & Data Overview





Key fields:

opened_at, resolved_at (timestamps)

Categorical:

impact, urgency, category, contact_type

Numeric:

sys_mod_count, time intervals…

Data Cleaning & Preparation

3

Data Import & Libraries

In this project, we used a variety of Python libraries to cover the full data science workflow — from data loading and exploration to modeling and evaluation.

- We used **Pandas** and **NumPy** for **efficient data handling** and manipulation.
- **Matplotlib** and **Seaborn** helped us **visualize the dataset** and extract meaningful patterns during EDA.
- **Scikit-learn** was used for **preprocessing**, **model training**, and evaluation of classical machine learning models.
- **XGBoost** and **LightGBM** provided advanced ensemble techniques for better accuracy.
- We used Keras (with TensorFlow backend) to implement a deep learning regression model for resolution time prediction.

Library	Purpose
`pandas`, `numpy`	Data loading, cleaning, manipulation
`matplotlib`, `seaborn`	Data visualization and exploratory analysis
`sklearn`	Data preprocessing, model building, evaluation
`xgboost`, `lightgbm`	Gradient boosting models for regression
`keras` / `tensorflow`	Deep learning model for regression task

Data Cleaning & Timestamp Parsing

One of the first critical steps was preparing the timestamps and eliminating unusable records to ensure data integrity.

- **Parsed Timestamps**: Converted `opened_at` and `resolved_at` strings into datetime objects.
- **Dropped Incomplete Rows**: Removed records missing either timestamp to ensure each incident had a full lifecycle.
- **Dropped Duplicates**: Removed duplicate rows to prevent biased modeling.
- **Pruned Irrelevant Columns**: Excluded fields like `caller_id`, `sys_created_by`, and free-text fields that do not contribute to prediction.

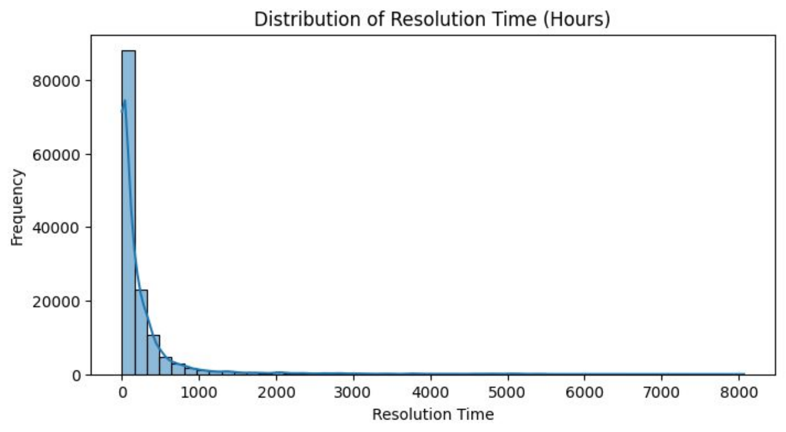
```python
# Convert datetime columns
date_cols = ['opened_at', 'resolved_at', 'closed_at']
for col in date_cols:
    df[col] = pd.to_datetime(df[col], errors='coerce')

# Remove rows with missing key timestamps
df = df.dropna(subset=['opened_at', 'resolved_at'])

# Create target variable: resolution time in hours
df['resolution_time_hours'] = (df['resolved_at'] - df['opened_at']).dt.total_seconds() / 3600


# Drop irrelevant columns (adjust if needed)
drop_cols = [
    'caller_id', 'opened_by', 'sys_created_by', 'sys_updated_by',
    'resolved_by', 'closed_at', 'sys_created_at', 'sys_updated_at'
]
df = df.drop(columns=drop_cols, errors='ignore')
```

Outlier Handling Strategy

Initially, we observed that the `resolution_time_hours` distribution was **heavily right-skewed**.

- We tried the standard **IQR rule**, but the results were not meaningful for our dataset's shape and business context.

- Instead of applying a blanket rule, we have implemented IQR rule in a **group-wise outlier removal** approach based on business-related features.
    a. opened_hour
    b. opened_dayofweek
    c. opened_weekend
    d. priority
    e. impact
    f. urgency
    g. day_half_day

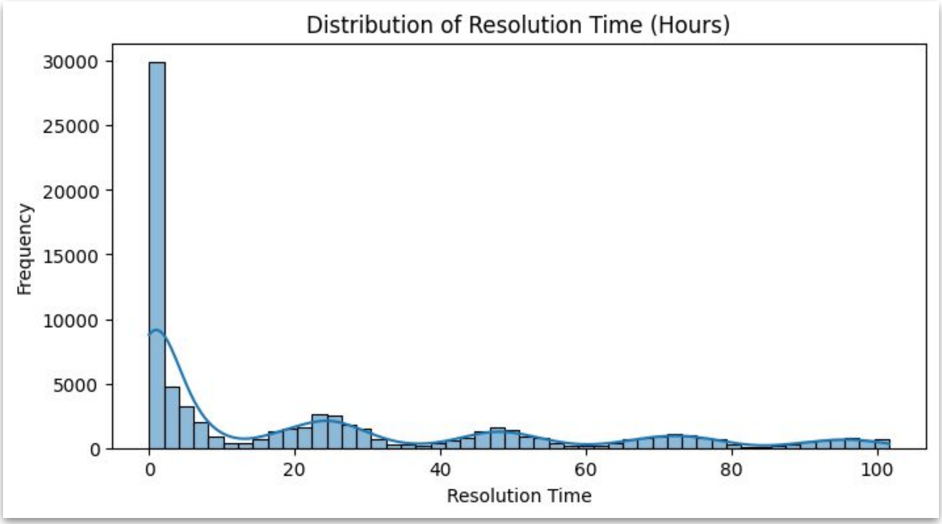



```
df = remove_outliers_grouped(df, 'day_half_day', 'resolution_time_hours')
df = remove_outliers_grouped(df, 'opened_hour', 'resolution_time_hours')
df = remove_outliers_grouped(df, 'opened_dayofweek', 'resolution_time_hours')
df = remove_outliers_grouped(df, 'opened_weekend', 'resolution_time_hours')
df = remove_outliers_grouped(df, 'priority', 'resolution_time_hours')
df = remove_outliers_grouped(df, 'impact', 'resolution_time_hours')
df = remove_outliers_grouped(df, 'urgency', 'resolution_time_hours')
```

Exploratory Data Analysis

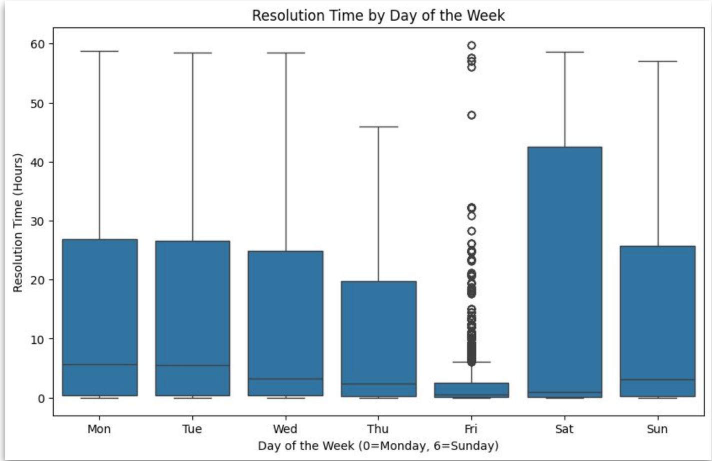
4

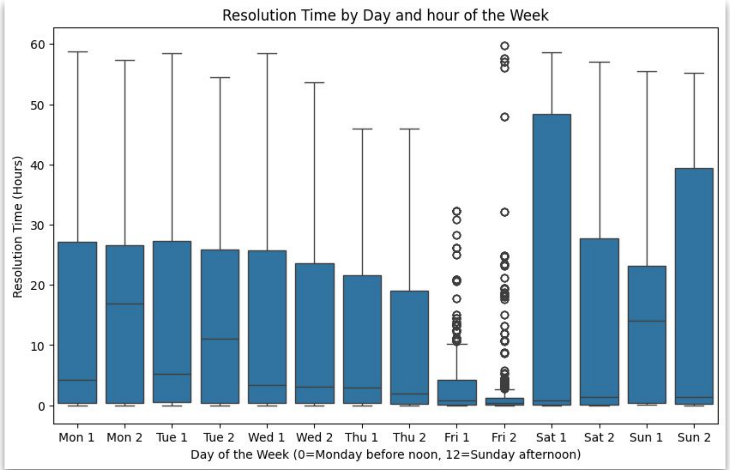
Exploratory Data Analysis - Resolution Time



- The histogram reveals a **strong right skew** in resolution times
- Most incidents are resolved **within the first few hours**.
- A long tail shows **a smaller group of incidents** that take significantly longer to resolve.
- This imbalance indicates **non-normality**, which affects model assumptions and evaluation.
- Guided our decision to apply **custom outlier removal** and careful preprocessing to avoid model distortion.
- The **slight peaks around every 20–24 hours** may suggest a weak seasonality **pattern** in resolution times, possibly reflecting **daily work cycles.**

Exploratory Data Analysis

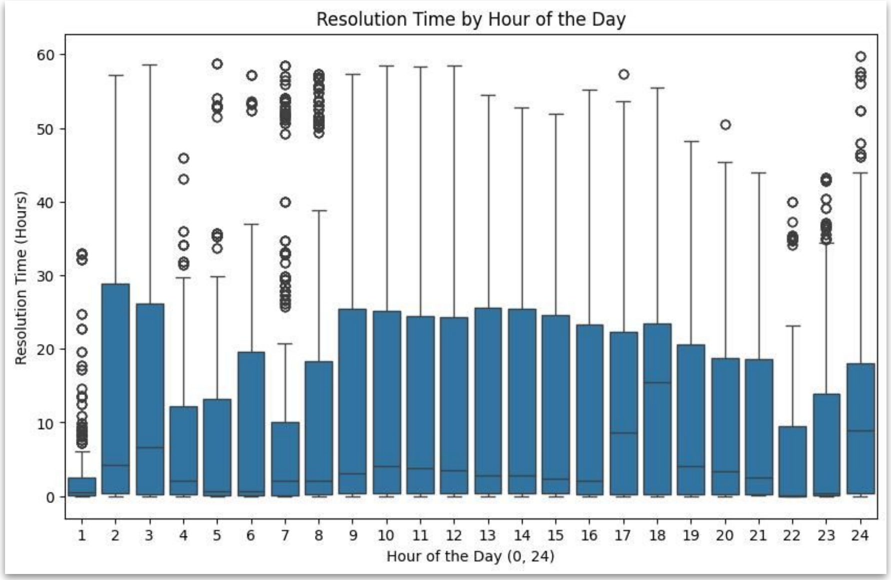




- Resolution times are **shortest on Fridays**, likely due to end-of-week urgency.
- **Saturdays show the highest mean** and variability, indicating weekend delays.
- Weekdays (Mon–Thu) have **similar patterns**, with moderate resolution times.
- Suggests that **day of the week significantly affects** how quickly incidents are resolved.

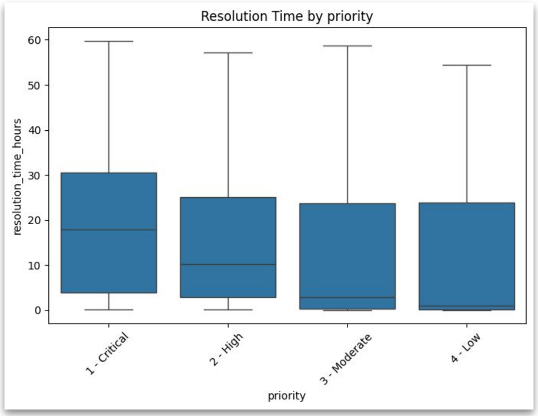
- **Resolution time** tends to be **shortest on Friday afternoons** — possibly due to pressure to close tasks before the weekend.
- **Sunday mornings** show the **highest median** and spread, indicating delays and fewer active staff.
- **Weekday afternoons** (Mon–Thu 1) generally **have longer resolution times than their mornings** counterparts (2).
- Highlights how both **day and time of day** affect **incident handling speed** — useful for staffing and scheduling insights.

Exploratory Data Analysis

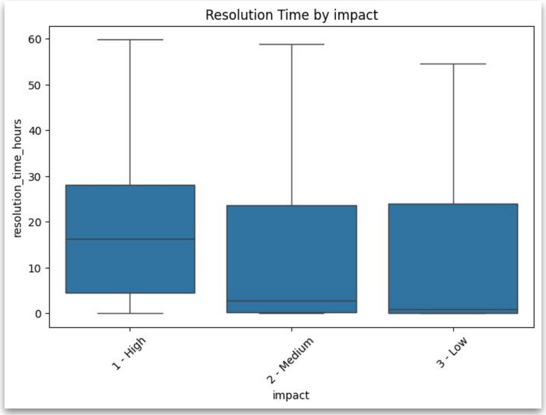


- Incidents created during **early morning hours** (1–3 AM) **show high variability** and frequent long resolution times.
- During **regular working hours** (9 AM–5 PM), resolution times **stabilize with lower medians** and more consistent patterns.
- A **spike in resolution time** appears **late at night** (11 PM–12 AM), likely due to limited support staff.
- **This pattern** emphasizes how **incident creation time impacts efficiency** and may help optimize staffing or SLA strategies.

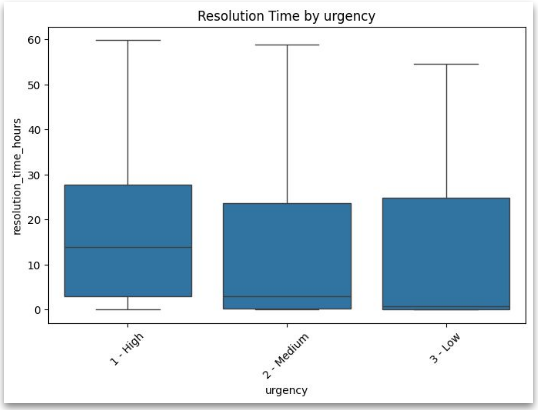
Exploratory Data Analysis



-Surprisingly, **higher-priority incidents** (tend to have **longer resolution times** than lower-priority ones.
-This may reflect the **complexity and severity** of critical cases rather than delays.
-Lower-priority tickets (e.g., *Low*, *Moderate*) are often resolved faster and with less variability.
-Indicates that **priority alone doesn't guarantee speed**, but rather signals the **difficulty of the issue**.

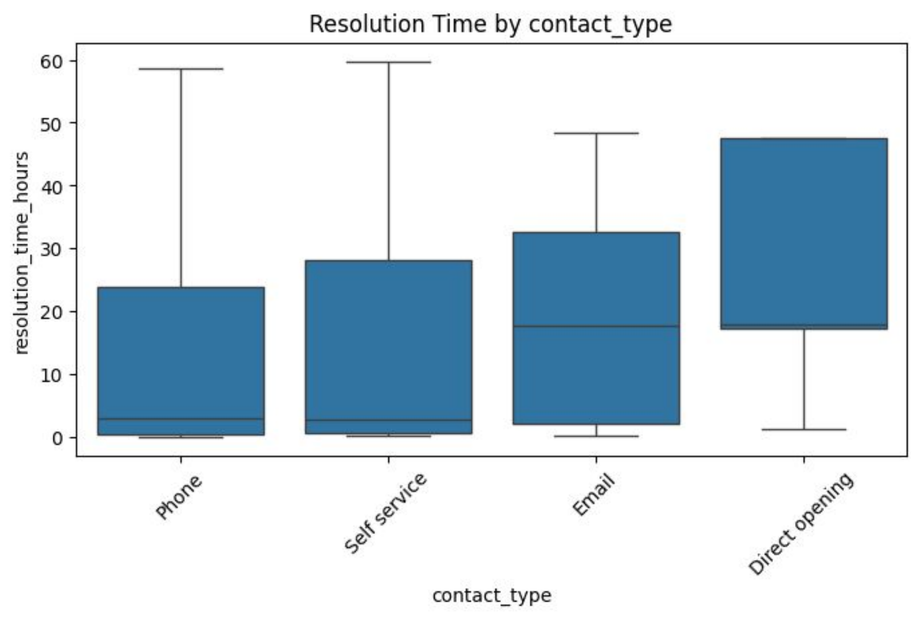


-Incidents with **high impact** generally take **longer to resolve**, showing higher medians and broader spread.
-**Medium and low impact** incidents are resolved **more quickly** and consistently.
-Resolution time still shows variability across all levels, but the trend is clear: higher impact → longer resolution.
-Confirms that **impact level is a strong predictor** of resolution time and a valuable feature for modeling.



-Higher urgency incidents tend to have **longer resolution times**, similar to the impact and priority trends.
-Despite urgency indicating importance, it may reflect the **complexity of critical cases**, not speed of resolution.
-Low and medium urgency incidents generally show **shorter and more consistent** resolution durations.
-Suggests urgency is a **key feature** to include for predicting resolution time — though not always inversely related to speed.

Exploratory Data Analysis



- Incidents reported via **email** and **direct opening** tend to have the **longest resolution times**.
- **Phone** and **self-service** contacts are resolved more quickly on average, with tighter distributions.
- The variability in resolution suggests some contact methods may **delay triaging or assignment**.
- Indicates that **how an incident is reported** can significantly influence response time — a valuable predictive feature.

Feature Engineering & Preprocessing

Feature Engineering: Target Variable Creation

```
# Create target variable: resolution time in hours
df['resolution_time_hours'] = (df['resolved_at'] - df['opened_at']).dt.total_seconds() / 3600
```

- Created a new column called `resolution_time_hours` — our prediction target.
- Calculated it as the **time difference between `resolved_at` and `opened_at`**, converted to hours.
- This gives us a **continuous numeric target** suitable for regression modeling.
- Ensures all records are labeled with **realistic and consistent resolution durations**, forming the foundation for model training.

Feature Engineering: Time-Based Features

```
[ ] # Time-based features for EDA (and later modeling)
    df['opened_dayofweek'] = df['opened_at'].dt.dayofweek
    df['opened_hour'] = df['opened_at'].dt.hour
    df['opened_weekend'] = df['opened_dayofweek'].isin([5, 6])
```

```
[ ] # Encode opened_hour into 'before_noon' and 'after_noon'
    df['opened_half_day'] = df['opened_hour'].apply(lambda x: 'before_noon' if x < 12 else 'after_noon')

    # Combine day of the week and half-day information
    df['day_half_day'] = df['opened_dayofweek'].astype(str) + '_' + df['opened_half_day']
```

- Extracted **`opened_dayofweek`** to capture the weekday (0 = Monday, 6 = Sunday) — useful for identifying weekday vs. weekend patterns.
- Created **`opened_hour`** to detect shifts and after-hours effects on resolution time.
- Added **`opened_weekend`** as a boolean feature to highlight incidents opened on Saturday or Sunday.
- Encoded **`opened_half_day`** as `before_noon` vs. `after_noon` to simplify the 24-hour timeline.
- Combined time and day into **`day_half_day`** (e.g., `Mon_before_noon`) for more granular time-context modeling.

Feature Engineering & Preprocessing

```
# Label encode categorical columns
categorical_cols = df_model.select_dtypes(include='object').columns
encoders = {}
for col in categorical_cols:
    le = LabelEncoder()
    df_model[col] = le.fit_transform(df_model[col])
    encoders[col] = le
```

- Applied **label encoding** to convert all object-type (categorical) columns into numeric format. This transformation allows machine learning algorithms to process non-numeric data effectively.
- For ordinal fields like **impact** and **urgency**, label encoding **preserves their inherent rank** (Low → Medium → High), which is meaningful for tree-based models.
- For nominal fields like **contact_type** or **category**, we later used **one-hot encoding** to avoid introducing false order and ensure equal treatment across categories.

Feature Engineering: Custom Ordinal Encoding Based on Medians

```
day_custom_mapping = {
    4: 5,  # Friday
    5: 5,  # Saturday
    6: 3,  # Sunday
    0: 2,  # Monday
    1: 0,  # Tuesday
    2: 1,  # Wednesday
    3: 3   # Thursday
}
```

```
# Define a custom mapping for each day and half-day combination
day_half_day_custom_mapping = {
    '0_before_noon': 6, '0_after_noon': 20,  # Monday
    '1_before_noon': 6, '1_after_noon': 17,    # Tuesday
    '2_before_noon': 3, '2_after_noon': 3,    # Wednesday
    '3_before_noon': 4, '3_after_noon': 14,    # Thursday
    '4_before_noon': 4, '4_after_noon': 3,  # Friday
    '5_before_noon': 9, '5_after_noon': 34,  # Saturday
    '6_before_noon': 6, '6_after_noon': 18   # Sunday
}
```

```
day_custom_mapping = {
    0: 1,
    1: 7,
    2: 7,
    3: 8,
    4: 1,
    5: 3,
    7: 3,
    8: 4,
    9: 5,
    10: 5,
    11: 5,
    12: 4,
    13: 19,
    14: 14,
    15: 19,
    16: 18,
    17: 20,
    18: 16,
    19: 14,
    20: 12,
    21: 0,
    22: 1,
    23: 10,
}
```

- Instead of default label encoding, we applied **custom mappings** to time-related features based on the **median resolution times** observed in the EDA.
- These mappings help the model learn **seasonal or behavioral patterns** tied to:
  - Day of the week
  - Hour of the day
  - Day–half-day combinations (e.g., `Friday_after_noon`)
- For example:
  - **Monday vs. Friday** may affect how quickly tickets are handled.
  - **Before noon vs. after noon** captures productivity cycles during the day.
- These rankings assign **more meaningful numeric values**, improving model accuracy compared to arbitrary label encodings.

Data Preprocessing for Modeling

```
# Train-test split
X = df_model.drop('resolution_time_hours', axis=1)
y = df_model['resolution_time_hours']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- Separated the features (**X**) from the target variable (**`resolution_time_hours`**).
- Split the dataset into **training (80%)** and **testing (20%)** sets using `train_test_split()`.
- Used a **random seed (`random_state=42`)** to ensure reproducibility of results.
- This allows us to **train the model on one portion** of the data and **evaluate its performance on unseen data**, simulating real-world prediction scenarios.

Machine Learning Models

6

Machine Learning Models

- **Linear Regression:** Baseline
  - Serves as a simple, interpretable benchmark for comparison.
- **K Nearest Neighbors (KNN):** k = 2
  - Captures local incident similarities and predicts based on the behavior of the most similar historical incidents.
- **Random Forest:** 100 trees for ensemble bagging stability.
  - Reduces overfitting by aggregating results across multiple decision trees.
- **XGBoost:** gradient boosting ensemble performance.
  - Sequentially corrects errors to enhance predictive accuracy.
- **Feedforward Neural Network:** 2 layers, ReLU for nonlinear patterns.
  - Models complex relationships through layered nonlinear transformations.

Machine Learning Models - Linear Regression

Linear Regression was used as a baseline model due to its simplicity and interpretability.

The model assumes a **linear relationship** between input features and the target variable (`resolution_time_hours`).

Evaluation metrics:

- **MAE (Mean Absolute Error): 11.01** → On average, the model is off by about 11 hours.
- **RMSE (Root Mean Squared Error): 14.14** → Penalizes larger errors more heavily.
- **R^2 (R-squared): 0.27** → The model explains only **27% of the variance** in resolution times.

```
# Linear Regression
from sklearn.linear_model import LinearRegression

lr_model = LinearRegression()
lr_model.fit(X_train, y_train)
y_pred_lr = lr_model.predict(X_test)

evaluate_model(y_test, y_pred_lr, "Linear Regression")
```

```
Linear Regression Performance:
  MAE:  11.01
  RMSE: 14.14
  R²:   0.27
```

This suggests that resolution time is likely **non-linear** and influenced by **more complex patterns** than linear regression can capture.

Machine Learning Models - K-Nearest Neighbors Regressor

KNN Regressor predicts a value based on the **average resolution times of the 'k' most similar cases** in the training set (here, k = 2).

It's a **non-parametric** model — it doesn't assume any specific data distribution, which helps capture local patterns. We tested multiple values for k, and **k=2 gave the best results**.

Evaluation Results:

- **MAE: 5.08** → Average error reduced to just over 5 hours.
- **RMSE: 10.58** → Still some large prediction errors, but better than linear regression.
- **R^2: 0.59** → Model explains **59% of the variance**, more than double what linear regression achieved.

Indicates that resolution time is influenced by **non-linear, instance-based patterns**, which KNN can better capture.

```
[ ] from sklearn.neighbors import KNeighborsRegressor

    knn_model = KNeighborsRegressor(n_neighbors=2)
    knn_model.fit(X_train, y_train)
    y_pred_knn = knn_model.predict(X_test)

    evaluate_model(y_test, y_pred_knn, "K-Nearest Neighbors")

    K-Nearest Neighbors Performance:
      MAE:  5.08
      RMSE: 10.58
      R²:   0.59
    ------------------------------
```

Machine Learning Models - Random Forest Regressor (Ensemble)

- Random Forest is an **ensemble model** that builds many decision trees and averages their predictions.
- It's excellent for **handling non-linear relationships** and **reducing overfitting** through bootstrapping and feature randomness.
- We used **100 trees** for a stable and robust result.

Model Performance:

- **MAE: 5.05** → Very low average error, on par with KNN.
- **RMSE: 8.93** → Better than both Linear and KNN models at controlling large errors.
- **R^2: 0.71** → Explains **71% of the variance** — the best result so far.

We'll fine-tune `n_estimators` and other hyperparameters later during the optimization phase to strike the right balance

```
from sklearn.ensemble import RandomForestRegressor

rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)

evaluate_model(y_test, y_pred_rf, "Random Forest")
```

```
Random Forest Performance:
  MAE:  5.05
  RMSE: 8.93
  R²:   0.71
------------------------------
```

Machine Learning Models - XGBoost Regressor (Ensemble)

- XGBoost is a **gradient boosting algorithm** known for speed and performance in structured/tabular data.
- It builds trees **sequentially**, where each tree learns from the previous one's errors — improving accuracy over time.
- We used **100 estimators** and the `reg:squarederror` objective for regression.

```
!pip install xgboost
from xgboost import XGBRegressor

xgb_model = XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42)
xgb_model.fit(X_train, y_train)
y_pred_xgb = xgb_model.predict(X_test)

evaluate_model(y_test, y_pred_xgb, "XGBoost")
xgb_model = XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42)
xgb_model.fit(X_train, y_train)
y_pred_xgb = xgb_model.predict(X_test)

evaluate_model(y_test, y_pred_xgb, "XGBoost")
```

```
Requirement already satisfied: xgboost in /usr/local/lib/python3.11/dist-packages (2.1.4)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from xgboo
Requirement already satisfied: nvidia-nccl-cu12 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from xgboo
XGBoost Performance:
  MAE:  7.59
  RMSE: 10.77
  R²:   0.58
------------------------------
```

Model Performance:

- **MAE: 7.59** → Higher error compared to Random Forest and KNN.
- **RMSE: 10.77** → Indicates more frequent larger deviations.
- **R^2: 0.58** → Explains **58% of variance**, lower than Random Forest (71%).

We will **fine-tune its parameters** (like `max_depth`, `learning_rate`, `n_estimators`, etc.) later during hyperparameter optimization.

Machine Learning Models - Deep Learning: Feedforward Neural Network (Keras)

- Built a simple **feedforward neural network** using two hidden layers with ReLU activation.
- Applied **feature scaling** (`StandardScaler`) — essential for neural networks to converge properly.
- Trained the model for **50 epochs**, using the **Adam optimizer** and mean squared error loss.

Model Performance:

- **MAE: 9.24** → Higher average error than tree-based models.
- **RMSE: 12.71** → Indicates presence of large prediction errors.
- **R^2: 0.41** → Explains **only 41% of the variance**, underperforming compared to Random Forest and KNN.

```
[ ] from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense
    from tensorflow.keras.optimizers import Adam
    from sklearn.preprocessing import StandardScaler

    # Deep learning needs scaled data
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Build the model
    dl_model = Sequential([
        Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
        Dense(32, activation='relu'),
        Dense(1)  # Output layer
    ])

    dl_model.compile(optimizer=Adam(learning_rate=0.001), loss='mse')

    # Train the model
    dl_model.fit(X_train_scaled, y_train, epochs=50, batch_size=32, verbose=0)

    # Predict
    y_pred_dl = dl_model.predict(X_test_scaled).flatten()

    evaluate_model(y_test, y_pred_dl, "Neural Network")
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
399/399 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
Neural Network Performance:
  MAE:  9.24
  RMSE: 12.71
  R²:   0.41
------------------------------
```

Model Optimization

7

Hyperparameter Tuning – Random Forest (Grid Search)

- Used **GridSearchCV** to test multiple combinations of hyperparameters and select the best-performing configuration.
- Parameters tuned:
  - `n_estimators` (number of trees)
  - `max_depth` (tree depth)
  - `min_samples_split`, `min_samples_leaf` (node splitting control)
- 3-fold cross-validation used with **negative mean squared error** as the scoring metric.

Tuned Model Performance:

- **MAE: 5.06** → Slight improvement over default model.
- **RMSE: 8.93** → Marginally better control of large errors.
- **R²: 0.71** → Same explained variance as the untuned version.

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor

# Define hyperparameter grid
param_grid_rf = {
    'n_estimators': [100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 3]
}

# Setup GridSearch
grid_rf = GridSearchCV(
    RandomForestRegressor(random_state=42),
    param_grid=param_grid_rf,
    cv=3,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)

# Run GridSearch
grid_rf.fit(X_train, y_train)
best_rf = grid_rf.best_estimator_
y_pred_best_rf = best_rf.predict(X_test)

evaluate_model(y_test, y_pred_best_rf, "Tuned Random Forest")
print("Best Random Forest Params:", grid_rf.best_params_)
```

```
Tuned Random Forest Performance:
  MAE:  5.06
  RMSE: 8.93
  R²:   0.71
------------------------------
```

Hyperparameter Tuning – XGBoost (Randomized Search)

- Used **RandomizedSearchCV** to explore a broad set of hyperparameters efficiently.
- Tuned the following key parameters:
  - `n_estimators`: number of boosting rounds
  - `max_depth`: tree depth
  - `learning_rate`: how quickly the model adapts
  - `subsample`: fraction of data used per tree (controls overfitting)
- Performed **10 iterations** with **3-fold cross-validation**, optimizing for **mean squared error**.

Tuned Model Performance:

- **MAE: 6.09** → Significant improvement over untuned XGBoost (previously 7.59)
- **RMSE: 9.11** → Handles large errors better
- **R^2: 0.70** → Nearly matches the performance of the tuned Random Forest

```python
from sklearn.model_selection import RandomizedSearchCV
from xgboost import XGBRegressor

param_dist_xgb = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.3],
    'subsample': [0.6, 0.8, 1.0]
}

random_search_xgb = RandomizedSearchCV(
    XGBRegressor(objective='reg:squarederror', random_state=42),
    param_distributions=param_dist_xgb,
    n_iter=10,
    cv=3,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)

random_search_xgb.fit(X_train, y_train)
best_xgb = random_search_xgb.best_estimator_
y_pred_best_xgb = best_xgb.predict(X_test)

evaluate_model(y_test, y_pred_best_xgb, "Tuned XGBoost")
print("Best XGBoost Params:", random_search_xgb.best_params_)
```

```
Tuned XGBoost Performance:
  MAE:   6.09
  RMSE:  9.11
  R²:    0.70
------------------------------
```

Model Comparison & Evaluation

8

Model Comparison Summary – Key Insights

	Model	MAE	RMSE	R^2
3	Tuned Random Forest	5.056976	8.930607	0.710034
2	Random Forest	5.048006	8.932982	0.709880
5	Tuned XGBoost	6.093580	9.106514	0.698499
1	K-Nearest Neighbors	5.075011	10.577193	0.593252
4	XGBoost	7.594918	10.767818	0.578458
6	Neural Network	9.241345	12.707736	0.412887
0	Linear Regression	11.006219	14.141106	0.272971

Tuned Random Forest achieved the **best overall performance** with the lowest RMSE (**8.93**) and highest R^2 (**0.71**).

Tuning improved both XGBoost and Random Forest, with XGBoost's R^2 jumping from **0.58** → **0.70** after optimization.

K-Nearest Neighbors performed surprisingly well (R^2 = **0.59**) despite its simplicity and minimal tuning.

Neural Network and Linear Regression had **weaker results**, likely due to:

- Neural network requiring more tuning/data
- Linear regression being too simple for complex, non-linear patterns

Tree-based models clearly **outperformed all others** — confirming their suitability for this tabular, structured dataset.

Conclusions & Next Steps

9

Conclusion

The goal of the project was to **predict incident resolution time (in hours)** using historical event log data from an IT service management system.

We explored a wide range of **machine learning models**, including linear, instance-based, ensemble, boosting, and deep learning approaches.

After thorough evaluation and hyperparameter tuning:

- **Tuned Random Forest** achieved the **best overall performance**.
- **XGBoost (tuned)** followed closely, proving strong after optimization.

Models confirmed that **temporal features** (e.g., day of week, hour, half-day) and **priority-related features** (urgency, impact) play a major role in resolution time.

Simple models like linear regression and deep learning underperformed, showing the importance of **model–data alignment**.

Final Thoughts & Future Improvements

💭 Final Thoughts

- Tree-based models, especially **Random Forest**, showed **robust performance** for predicting incident resolution time.
- Feature engineering based on **timestamp decomposition** and **domain-specific rankings** significantly improved model accuracy.
- Simpler models struggled due to the **non-linear and context-sensitive nature** of incident resolution behavior.

🚀 Future Improvements

- 🔧 **Further Hyperparameter Tuning**: Explore deeper trees, feature selection, and boosting-specific strategies (e.g., early stopping).
- 📅 **Time-Aware Validation**: Use **time series split** instead of random split to better simulate real-world forecasting.
- 🧠 **Model Explainability**: Apply SHAP or LIME to understand which features influence resolution time the most.
- 📈 **Live Feedback Loop**: Integrate real-time predictions into incident management systems to help allocate resources proactively.
- 🧩 **Feature Expansion**: Include more variables like team workload, ticket source details, or escalation history.

Thank you