

# **AI-AUGMENTED WORKFLOW FOR DATA ANALYSTS – PART 3**

Cursor-based automation example and summary of agentic approach

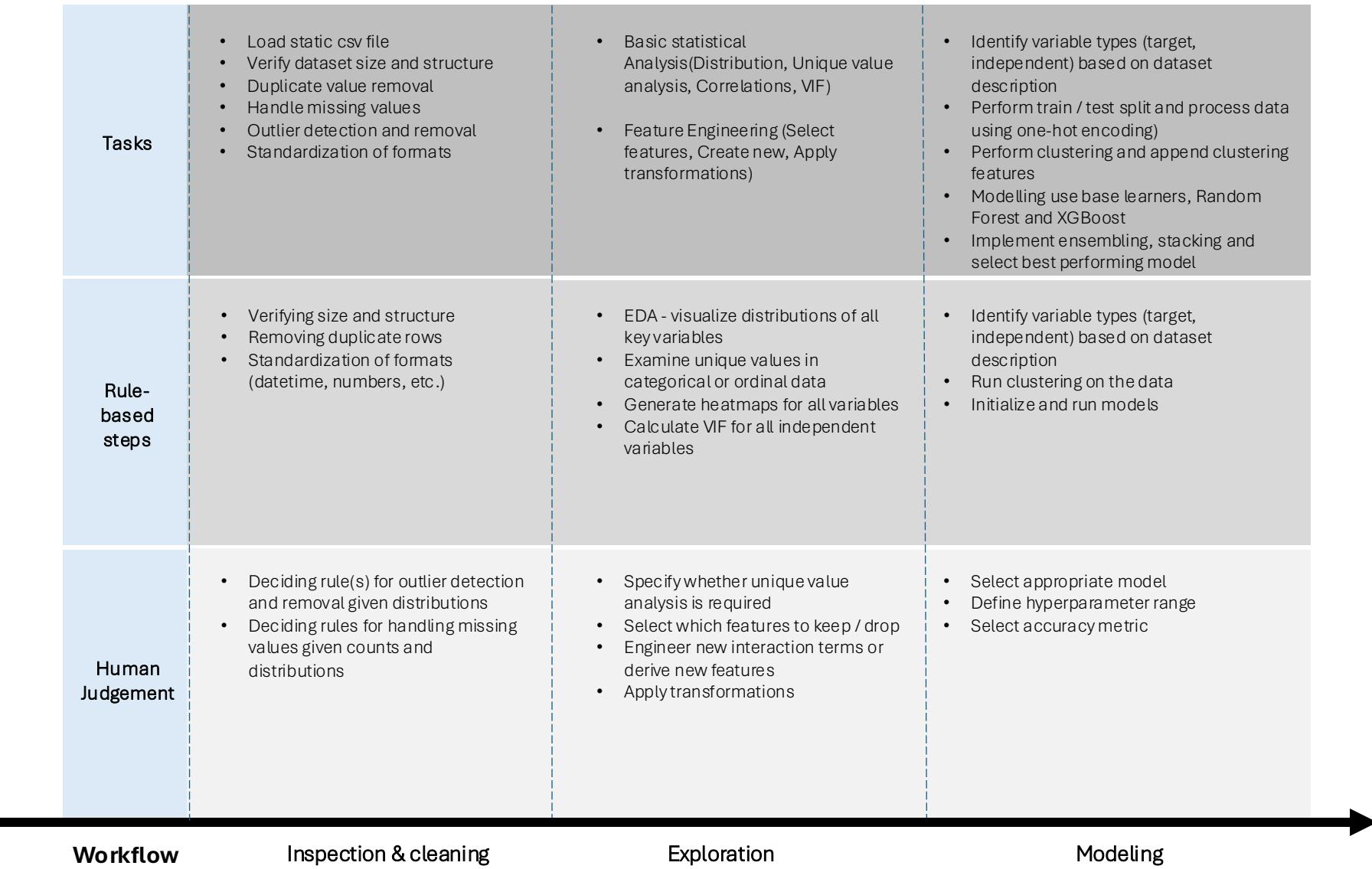
*Prepared by - Seung-Mi Jeon, Hrishikesh Bhatt*

# MILESTONE OVERVIEW

## Progress since last milestone

1. Refined the prompt frames and ran an entire predictive analysis workflow on Cursor using the NYC Bikeshare dataset
  - Prediction task: Predicted Gender of riders using 3 independent variables
  - Achieved accuracy of ~77.9% across various models
2. Created an agent using LangChain and OpenAI API to perform entire workflow through a web-based UI
  - Reorganize code templates with LLM placeholder
  - Explore 3 structures and Implement using LangChain and Fast API endpoints

# REVISED WORKFLOW (CUSTOMIZED FOR PREDICTIVE MODELLING)



# UPDATED PROMPTING FORMAT

## Modelling Workflow Prompt

You are a highly experienced data analyst responsible for designing and executing the modeling phase of a data analysis pipeline. Your role is to perform predictive modelling based on the logic provided below in this prompt.

.....

While undertaking tasks 1-5, use the additional natural language inputs provided in the variables `modelling\_context` and `dataset\_description`, which are given as strings. The information is contained in the file "context.txt". The dataset to use is 'dataset\_cleaned\_updated.csv'

..... The `target variable` and `independent\_variables` are also mentioned in "context.txt".

## `Task` 4: Model Execution & Optimization

**\*\*Base Learners + Grid-Search Tuning (NO nested CV)\*\***

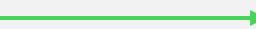
**•\*\*Classification\*\*:** `LogisticRegression`, `RandomForestClassifier`, `XGBClassifier`

**•\*\*Regression\*\*:** `LinearRegression`, `RandomForestRegressor`, `XGBRegressor`

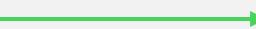
•Provide concise, sensible `param\_grid` for each; use \*\* `GridSearchCV` (cv=5)`\*\* to select best parameters.

## —— OUTPUT FORMAT ——

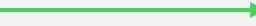
Return the fully commented Python code (.py script). No narrative or markdown outside code blocks. At the end of the script, compile all model names and their evaluation scores into a Python dictionary, and output it in JSON format."



Guidance to the model including role and main task



Calling back to `modelling context`, `dataset description`, `target variable` and `independent variables` (mentioned in context.txt document)



Provide decision making guidance criteria for each step

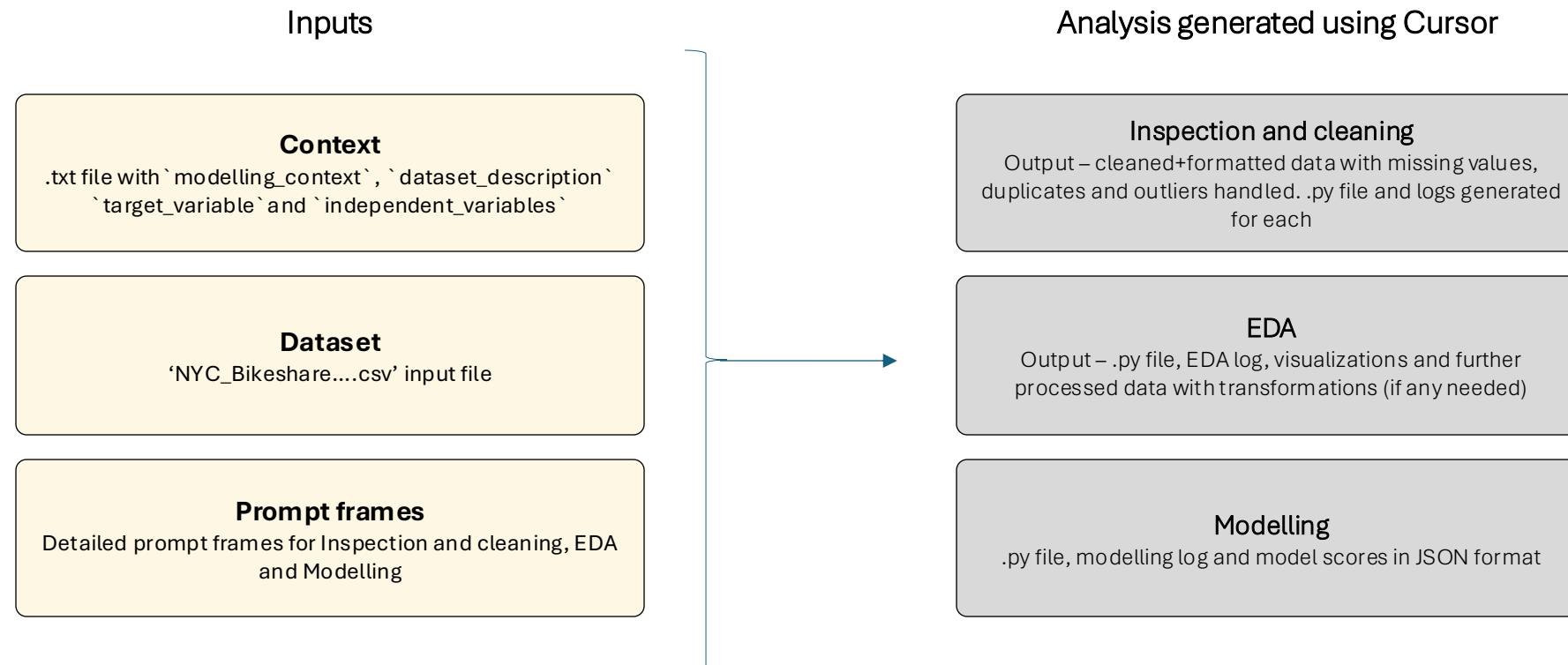


Provide clear output format and generate modelling comparison

# PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

## Modelling Problem Overview

- Experiment with **NYC-BikeShare-2015-2017-combined.csv**
- csv file (17 columns x 735503 rows)
- Contains anonymized trip information from NYC Bikeshare system from Jan 2015-Jun 2017 (Each row corresponds to a trip)
- Task: Predict 'Gender' of the rider based on 'Start Time', 'Trip\_Duration\_in\_min' and 'User Type' (Subscriber or regular User)



# PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

## Classification task results

- Best F-1 score of **0.7797** from XGBoost shows strong performance using prompt frames alone
- Results are robust, particularly given the paucity of features

	Model Type	Result (Micro-F1 score)
Base Models	Logistic Regression	0.7793
	Random Forest Classifier	0.7793
	XGBClassifier	0.7797
	Ensemble Using simple averaging	0.7793
	Stacked Model	0.7786

# PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

## Inspection and Cleaning (1/2)

The screenshot shows a Jupyter Notebook interface with several tabs open. The current tab is 'ic\_step1.py' (version 2). The code in the cell is as follows:

```
import pandas as pd
import numpy as np
import os
from datetime import datetime

# Create output directory if it doesn't exist
os.makedirs('ic_step1_output', exist_ok=True)
#L to chat, %K to generate
# Read the dataset
print("Reading the dataset...")
df = pd.read_csv('NYC-BikeShare-2015-2017-cleaned.csv')

# Initialize a log file for tracking changes
log_file = open('ic_step1_output/data_cleaning_log.txt', 'w')
log_file.write("Data Cleaning Log - Step 1\n")
log_file.write("=" * 50 + "\n\n")

# 1. Basic Dataset Information
log_file.write("1. Basic Dataset Information\n")
log_file.write("-" * 30 + "\n")
log_file.write(f"Total rows: {len(df)}\n")
log_file.write(f"Total columns: {len(df.columns)}\n\n")

# 2. Column Information
log_file.write("2. Column Information\n")
log_file.write("-" * 30 + "\n")
column_info = pd.DataFrame({
    'Data Type': df.dtypes,
    'Non-Null Count': df.count(),
    'Null Count': df.isnull().sum(),
    'Null Percentage': (df.isnull().sum() / len(df)).round(2)
})
log_file.write(column_info.to_string())
log_file.write("\n\n")

# 3. Data Format Validation
log_file.write("3. Data Format Validation\n")
log_file.write("-" * 30 + "\n")

# Check for whitespace in string columns
string_columns = df.select_dtypes(include=['object']).columns
for col in string_columns:
    if df[col].str.strip().ne(df[col]).any():
        log_file.write(f"Found whitespace issues in column: {col}\n")
        df[col] = df[col].str.strip()

# 4. Data Type Conversion
# 5. Missing Value Imputation
# 6. Outlier Detection and Treatment
# 7. Duplicate Removal
# 8. Data Transformation and Feature Engineering
```

To the right of the code editor is a 'Data Inspection and Cleaning Process' window. This window contains a 'Add context' section with instructions for the data analyst:

- You are a highly experienced data analyst who is inspecting and cleaning the data for further processing. Do not ask for user input using input() or any Python interactive function. Instead, for any 'human judgment' required, clearly state the question in the chat interface.
- You should generate a python file for each 'Task' (or step) in the process. Ensure that each generated file is clearly named (e.g., ic\_step1.py, ic\_step2.py, etc.) and output files from each step are located on the folder named 'ic\_step1\_output'. Please include descriptive comments above each code block explaining what it does. After generating the code, confirm that the file (e.g., ic\_step1.py) has been created.

Below these instructions, a message states: "There are 4 steps for inspection and cleaning. Where the step says 'human'".

At the bottom of the window, there is a terminal-like interface showing command-line history and a prompt for a new chat.

.py files for each of the 4 tasks –

Loading and studying structure, removing duplicates, handling missing values, handling outliers

Comprehensive prompt frame fed into the window (visual truncated)

# PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

## Inspection and Cleaning (2/2)

The screenshot shows a Jupyter Notebook interface with several files listed in the left sidebar:

- 3\_CURSOR - ETB FINAL PROJECT
- > Cursor Screenshots
- > eda\_output
- > ic\_step1\_output (highlighted by a red box)
- > ic\_step2\_output
- > ic\_step3\_output
- > ic\_step4\_output
- = context.txt
- dataset\_cleaned\_updated.csv
- dataset\_cleaned.csv
- eda.py
- ic\_step1.py 2
- ic\_step2.py 2
- ic\_step3.py 4
- ic\_step4.py 5
- { model\_scores.json
- = modeling.log
- eda.py
- NYC-BikeShare-2015-2017-cleaned...
- > NOTEPADS
- > OUTLINE
- > TIMELINE

The main area displays the content of `ic_step1.py`:

```
import pandas as pd
import numpy as np
import os
from datetime import datetime

# Create output directory if it doesn't exist
os.makedirs('ic_step1_output', exist_ok=True)

# Read the dataset
print("Reading the dataset...")
df = pd.read_csv('NYC-BikeShare-2015-2017-cleaned.csv')

# Initialize a log file for tracking changes
log_file = open('ic_step1_output/data_cleaning_log.txt', 'w')
log_file.write("Data Cleaning Log - Step 1\n")
log_file.write("=" * 50 + "\n")

# 1. Basic Dataset Information
log_file.write("1. Basic Dataset Information\n")
log_file.write("-" * 30 + "\n")
log_file.write(f"Total rows: {len(df)}\n")
log_file.write(f"Total columns: {len(df.columns)}\n\n")

# 2. Column Information
log_file.write("2. Column Information\n")
log_file.write("-" * 30 + "\n")
column_info = pd.DataFrame({
    'Data Type': df.dtypes,
    'Non-Null Count': df.count(),
    'Null Count': df.isnull().sum(),
    'Null Percentage': (df.isnull().sum() / len(df)).round(2)
})
log_file.write(column_info.to_string())
log_file.write("\n\n")

# 3. Data Format Validation
log_file.write("3. Data Format Validation\n")
log_file.write("-" * 30 + "\n")

# Check for whitespace in string columns
string_columns = df.select_dtypes(include=['object']).columns
for col in string_columns:
    if df[col].str.strip().ne(df[col]).any():
        log_file.write(f"Found whitespace issues in column: {col}\n")
        df[col] = df[col].str.strip()

# 4. Data Cleaning
log_file.write("4. Data Cleaning\n")
log_file.write("-" * 30 + "\n")
log_file.write("Handling outliers...\n")
log_file.write("-" * 30 + "\n")
```

A red box highlights the section from line 1 to line 17, which corresponds to the first two sections of the log file.

The right side of the interface shows a "Data Inspection and Cleaning Process" panel with a red border, containing the following text:

Based on the outlier analysis, we've found several important insights:

1. Trip Duration and Trip\_Duration\_in\_min:
  - About 9% of trips are outliers
  - Some trips are extremely long (capped at 1269.5 seconds and 21.5 minutes respectively)
  - Some negative durations were found (which shouldn't be possible)
2. Station IDs:
  - About 11-12% of station IDs are outliers
  - This suggests some stations are used much more frequently than others
3. Geographic Coordinates:
  - 2-5% of latitude/longitude values are outliers
  - Some coordinates are outside the expected NYC area
4. Bike IDs:
  - About 22% of bike IDs are outliers
  - This suggests some bikes are used much more frequently than others
5. Birth Year:
  - About 4% of birth years are outliers
  - Values are capped between 1958 and 2002
6. Gender:
  - About 28% of gender values are outliers
  - Values are capped at 1 (suggesting a binary encoding)

The script has handled these outliers by:

1. Removing outliers for columns with < 1% outliers
2. Capping outliers at IQR boundaries for columns with > 1% outliers

The final cleaned dataset has been saved as `'dataset_step4_cleaned.csv'`.

At the bottom, a status bar indicates: Edited 6 files +742-394, Add context, Plan, search, build anything, Agent, Auto, Start a new chat for better results. New chat.

Separate output log  
and visuals created in  
same directory

Additional insights generated in  
the chat window to summarize  
Inspection and Cleaning  
process

# PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

## Exploration (1/3)

The screenshot shows a Jupyter Notebook environment with the following components:

- File Explorer:** On the left, it shows a file tree for the project. A red box highlights the 'eda\_output' directory, and a red arrow points down from it to the text "Separate output log and visuals created in same directory".
- Code Editor:** The main area displays the 'eda.py' script. A red box highlights the first few lines of the script, and a red arrow points down from it to the text "Separate output log and visuals created in same directory".
- Data Inspection and Cleaning Process:** This is a separate window titled "Data Inspection and Cleaning Process" with the following content:
  - @ Add context
  - You are a highly experienced data analyst conducting exploratory data analysis. Do not ask for user input using input() or any Python interactive function. Instead, for any 'human judgment' required, clearly state the question in the chat interface.
  - You should generate a python code file with your analysis. After generating the code, confirm that the file (e.g., eda.py) has been created.
  - There are 4 steps for EDA. Only where 'human judgement' is mentioned, wait for the user's natural language response in the chat, and then proceed to generate the corresponding code in the python file. Proceed to the next step after a valid response is received from the chat. If 'human judgement' is not mentioned, execute your code and proceed to the next step.
- Code Preview:** Below the main code editor, there is a preview of the generated code, which is truncated.
- Terminal:** At the bottom, there is a terminal window showing the command "python eda.py" and its output.

Separate prompt frame for EDA (visual truncated)

# PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

## Exploration (2/3)

The screenshot shows a Jupyter Notebook environment with a sidebar containing project files like 'eda.py', 'ic\_step1.py', etc., and a main area with code and an AI-generated report.

**Code:**

```
eda.py 5 ×
eda.py > analyze_trip_duration
1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5 from scipy import stats
6 import os
7 from datetime import datetime
8
9 # Create output directory if it doesn't exist
10 output_dir = 'eda_output'
11 if not os.path.exists(output_dir):
12     os.makedirs(output_dir)
13
14 # Initialize log file
15 log_file = open(os.path.join(output_dir, 'eda_analysis_log.txt'), 'w')
16
17 def log_message(message):
18     print(message)
19     log_file.write(message + '\n')
20
21 def analyze_gender(df):
22     """Analyze gender distribution."""
23     log_message("\nAnalyzing Gender Distribution")
24
25     # Value counts and proportions
26     value_counts = df['Gender'].value_counts()
27     value_props = df['Gender'].value_counts(normalize=True)
28
29     log_message(f"Number of unique values: {len(value_counts)}")
30     log_message("\nGender distribution:")
31     for value, count in value_counts.items():
32         log_message(f"Gender {value}: {count} ({value_props[value]:.2%})")
33
34     # Create visualization
35     plt.figure(figsize=(10, 6))
36     sns.barplot(x=value_counts.index, y=value_counts.values)
37     plt.title('Gender Distribution')
38     plt.xlabel('Gender')
39     plt.ylabel('Count')
40     plt.tight_layout()
41     plt.savefig(os.path.join(output_dir, 'gender_analysis.png'))
42     plt.close()
43
44 def analyze_trip_duration(df):
45     """Analyze trip duration distribution."""
46
```

**Data Inspection and Cleaning Process**

Based on the analysis, we have several types of unusual variables that need handling:

- Trip Duration Variables:**
  - Trip Duration (seconds) and Trip\_Duration\_in\_min (minutes) are both numeric but have been flagged as categorical due to the unique values. These should remain numeric as they represent continuous time measurements.
- Station Information:**
  - Start Station ID and End Station ID: These are categorical variables that happen to be numeric. They should be treated as categorical.
  - Start Station Latitude/Longitude and End Station Latitude/Longitude: These are geographic coordinates and should remain numeric.
- Bike ID:**
  - This is a categorical variable (bike identifier) that happens to be numeric. It should be treated as categorical.
- Demographic Variables:**
  - Birth Year : This is a numeric variable representing years and should remain numeric.
  - Gender : This is a categorical variable (0, 1, 2) and should be treated as categorical.
- Time Variables:**
  - Start Hour : This is a categorical variable representing hours of the day (0-23) and should be treated as categorical.
- Imputation Flags:**
  - All columns ending with \_imputed are binary flags (0/1) indicating whether the original value was imputed. These should be treated as categorical.

Would you like me to proceed with these categorizations and continue

Ln 90, Col 20 Spaces: 4 UTF-8 LF Python 3.12.5 64-bit Cursor Tab

LLM Flagged several “unusual” variables that need handling:

- Mostly categorical variables flagged as numeric and vice-versa
- Interestingly the LLM made some errors too
  - it used birth year as a numeric variable while data transformation should have been suggested

# PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

## Exploration (3/3)

The screenshot shows a Jupyter Notebook interface with several tabs and sections:

- Left Sidebar:** Shows a file tree for "3\_CURSOR - ETB FINAL PROJECT" containing files like "Cursor Screenshots", "eda\_output", "ic\_step1\_output", etc.
- Code Cell:** The main code cell contains Python code for data analysis, including imports for pandas, numpy, matplotlib, seaborn, and stats, along with functions for analyzing gender distribution and trip duration.
- Output Section:** A "Data Inspection and Cleaning Process" section displays the results of the script's execution. It includes:
  - A message: "The script has completed successfully! Here's what was done:"
  - 1. Gender Analysis:**
    - Gender 1: 242,904 users (71.52%)
    - Gender 2: 69,197 users (20.37%)
    - Gender 0: 27,519 users (8.10%)
  - 2. Trip Duration Analysis:**
    - Mean: 8.43 minutes
    - Median: 6.00 minutes
    - Range: 1-21.5 minutes
    - No outliers detected using IQR method
  - 3. Start Time Analysis:**
    - Peak hours identified:
      - Hour 8: 40,472 trips
      - Hour 18: 33,515 trips
      - Hour 17: 30,344 trips
  - 4. Dataset Updates:**
    - Added 'is\_outlier' column to mark any outliers in trip duration
    - Added 'Start Hour' column for temporal analysis
    - Saved as 'dataset\_cleaned\_updated.csv'

The output also mentions visualizations created in the 'eda\_output' directory and asks if the user wants to save the updated dataset.

- Bottom Status Bar:** Shows "Ln 90, Col 20" and other system information.

Analysis summary generated in chat window and output with transformations saved in directory

LLM needed to be asked to focus on `target\_variable` and `independent\_variable` only

## PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

Modelling (1/3)

The screenshot shows a Jupyter Notebook environment with several open cells and a sidebar panel.

**Left Sidebar:**

- Project tree: 3\_CURSOR - ETB FINAL PROJECT
- Files: Cursor Screenshots, eda\_output, ic\_step1\_output, ic\_step2\_output, ic\_step3\_output, ic\_step4\_output, context.txt, dataset\_cleaned\_updated.csv, dataset\_cleaned.csv, eda.py, ic\_step1.py, ic\_step2.py, ic\_step3.py, ic\_step4.py, model\_scores.json, modeling.log, modeling.py, NYC-BikeShare-2015-2017-cleaned...

**Main Area (Code Cell 8):**

```
modeling.py 8 X
modeling.py > train_models
1 import pandas as pd
2 import numpy as np
3 from sklearn.model_selection import train_test_split, GridSearchCV
4 from sklearn.preprocessing import OneHotEncoder, StandardScaler
5 from sklearn.linear_model import LogisticRegression
6 from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
7 from xgboost import XGBClassifier
8 from sklearn.metrics import f1_score
9 import json
10 import logging
11 from datetime import datetime
12
13 # Set up logging
14 logging.basicConfig(
15     filename='modeling.log',
16     level=logging.INFO,
17     format='%(asctime)s - %(levelname)s - %(message)s'
18 )
19
20 def load_and_prepare_data():
21     """Task 1: Load data and prepare variables"""
22     try:
23         print("Loading dataset...")
24         # Load the dataset
25         df = pd.read_csv('dataset_cleaned_updated.csv')
26         print(f"Dataset loaded with shape: {df.shape}")
27
28         print("Converting Start Time to datetime...")
29         # Convert Start Time to datetime
30         df['Start Time'] = pd.to_datetime(df['Start Time'])
31
32         print("Extracting temporal features...")
33         # Extract temporal features
34         df['Start Hour'] = df['Start Time'].dt.hour
35         df['Start Day'] = df['Start Time'].dt.day
36         df['Start Month'] = df['Start Time'].dt.month
37         df['Start Weekday'] = df['Start Time'].dt.weekday
38
39         # Define target and features
40         target = 'Gender'
41         features = ['Start Hour', 'Start Day', 'Start Month', 'Start Weekday',
42                     'Trip Duration in_min', 'User Type']
43
44         print("Separating features and target...")
45         # Separate X and y
```

**Data Inspection and Cleaning Process Panel:**

Modeling Context:

You are a highly experienced data analyst responsible for designing and executing the modeling phase of a data analysis pipeline. Your role is to perform predictive modelling based on the logic provided below in this prompt.

While undertaking tasks 1-5, use the additional natural language inputs provided in the variables 'modelling\_context' and 'dataset\_description', which are given as strings. The information is contained in the file "context.txt". The dataset to use is 'dataset\_cleaned\_updated.csv'. Generate a \*\*complete, runnable Python script\*\* (.py file with no additional user input required) that performs predictive modelling on a pre-loaded pandas DataFrame called \*\*df\*\* whose target column is \*\*\*{target\_variable}\*\*\* and independent variables are \*\*\*{independent\_variables}\*\*\*. The 'target variable' and 'independent variables' are defined in the 'modelling\_context' variable.

Agent Auto

I'll help you create a comprehensive modeling script. First, let me check the context file to understand the modeling requirements:

Read context.txt

Now I'll create a comprehensive modeling script based on the requirements:

modeling.py +256 -1 ✓

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.metrics import f1_score
```

Edited 6 files +742 -394

Plan, search, build anything

Agent Auto

Start a new chat for better results. New chat

Separate output log  
and json file with  
results created in same  
directory

Separate prompt frame for  
Modelling  
(visual truncated)

# PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

## Modelling (2/3)

The screenshot shows a terminal window with several tabs open. The tabs include 'modeling.py 8', 'modeling.log X', and 'model\_scores.json'. The 'modeling.log' tab displays log entries from a script run on May 4, 2025. The log entries show errors during clustering and successful training of various classifiers (LogisticRegression, RandomForestClassifier, XGBClassifier) with F1 scores around 0.779. The 'model\_scores.json' tab shows a JSON object with model names and their F1 scores. To the right of the terminal, a 'Data Inspection and Cleaning Process' window is open, displaying a summary of the process. A red box highlights the summary text, and a red arrow points from this box to a caption below.

```
modeling.log -- 3_cursor - ETB final project
```

modeling.py 8 modeling.log model\_scores.json

modeling.log

```
1 2025-05-04 14:35:10,309 - ERROR - Error in clustering: OneHotEncoder.__init__() got an unexpected keyword argument 'drop'
2 2025-05-04 14:35:10,309 - ERROR - Error in main execution: OneHotEncoder.__init__() got an unexpected keyword argument 'drop'
3 2025-05-04 14:35:51,965 - ERROR - Error in clustering: 'NoneType' object has no attribute '__init__'
4 2025-05-04 14:35:51,965 - ERROR - Error in main execution: 'NoneType' object has no attribute '__init__'
5 2025-05-04 14:37:03,808 - ERROR - Error in clustering: 'NoneType' object has no attribute '__init__'
6 2025-05-04 14:37:03,808 - ERROR - Error in main execution: 'NoneType' object has no attribute '__init__'
7 2025-05-04 14:37:46,957 - ERROR - Error in clustering: 'NoneType' object has no attribute '__init__'
8 2025-05-04 14:37:46,957 - ERROR - Error in main execution: 'NoneType' object has no attribute '__init__'
9 2025-05-04 14:38:31,368 - INFO - LogisticRegression trained successfully with score: 0.7793
10 2025-05-04 14:38:31,369 - INFO - Best parameters: {'C': 0.1, 'max_iter': 1000}
11 2025-05-04 14:41:19,571 - INFO - LogisticRegression trained successfully with score: 0.7793
12 2025-05-04 14:41:19,572 - INFO - Best parameters: {'C': 0.1, 'max_iter': 1000}
13 2025-05-04 14:47:14,891 - INFO - RandomForestClassifier trained successfully with score: 0.7793
14 2025-05-04 14:47:14,892 - INFO - Best parameters: {'max_depth': 10, 'n_estimators': 100}
15 2025-05-04 14:47:55,324 - INFO - XGBClassifier trained successfully with score: 0.779665508
16 2025-05-04 14:47:55,325 - INFO - Best parameters: {'max_depth': 5, 'n_estimators': 100}
17
```

Data Inspection and Cleaning Process

Great! The script has completed successfully. Here's a summary of what happened:

1. Data Loading and Preparation:
  - Successfully loaded the dataset with 339,620 rows and 36 columns
  - Extracted temporal features from the Start Time
  - Prepared 6 features for modeling
2. Feature Processing:
  - Scaled numeric features
  - One-hot encoded categorical features
  - Final processed feature set has 7 columns
3. Model Training and Evaluation:
  - Split data into training (271,696 samples) and test (67,924 samples) sets
  - Trained 5 different models:
    - XGBClassifier: 0.7797 (best performance)
    - LogisticRegression: 0.7793
    - Simple Ensemble: 0.7793
    - RandomForestClassifier: 0.7793
    - Stacked Model: 0.7786
4. Results:
  - All models performed similarly well, with F1 scores around 0.779
  - XGBClassifier performed slightly better than the others
  - The results have been saved to 'model\_scores.json'

Would you like to:

1. Analyze the model results in more detail?
2. Try different feature combinations?
3. Adjust the model parameters?

Edited 6 files +742 -394

@ Add context

Plan, search, build anything

Agent Auto

Start a new chat for better results. New chat

Ln 1, Col 1 Spaces: 4 UTF-8 LF Log Cursor Tab

Modelling log generated in directory and summarized in chat window

# PREDICTIVE ANALYSIS USING PROMPT FRAMES ON CURSOR

## Modelling (3/3)

The screenshot shows a code editor interface with a dark theme. On the left is a sidebar with project files: 3\_CURSOR - ETB FINAL PROJECT, modeling.py, modeling.log, model\_scores.json, context.txt, dataset\_cleaned\_updated.csv, dataset\_cleaned.csv, eda.py, ic\_step1.py, ic\_step2.py, ic\_step3.py, ic\_step4.py, and NYC-BikeShare-2015-2017-cleaned... The main area has tabs for modeling.py, modeling.log, and model\_scores.json, with model\_scores.json currently active. The content of model\_scores.json is:

```
1 "XGBClassifier": 0.7796655085095107,  
2 "LogisticRegression": 0.779326894764737,  
3 "Simple Ensemble": 0.779326894764737,  
4 "RandomForestClassifier": 0.7792974500912785,  
5 "Stacked Model": 0.7785613332548142
```

To the right of the code editor is a panel titled "Data Inspection and Cleaning Process". It contains a sub-section titled "I'll analyze the model results in more detail. Provide your analysis only in this chat". The analysis is presented in two parts:

- Overall Model Performance:**
  - All models achieved remarkably similar performance, with F1 scores ranging from 0.7786 to 0.779
  - This narrow range (0.0011 difference) suggests that the models are capturing similar patterns in the data
  - The F1 score of ~0.779 indicates that the models are performing significantly better than random chance (which would be 0.333 for a 3-class problem)
- Individual Model Analysis:**
  - XGBClassifier (0.7797):**
    - Best performing model
    - Used relatively simple parameters (`max_depth=5, n_estimators=100`)
    - This suggests that the relationship between features and target is not highly complex
  - LogisticRegression (0.7793):**
    - Performed almost as well as XGBClassifier
    - Best parameters: `C=0.1`, indicating preference for stronger regularization
    - This suggests that the decision boundaries are relatively linear
  - RandomForestClassifier (0.7793):**
    - Matched LogisticRegression's performance
    - Used moderate depth (`max_depth=10`) and 100 trees
    - The similar performance to LogisticRegression suggests that the feature interactions are not highly complex

Annotations with red boxes and arrows highlight the following:

- A red box surrounds the title "model\_scores.json" in the tab bar, with an arrow pointing to the text "Model scores summarized in json file".
- A large red box surrounds the entire analysis section in the panel, with an arrow pointing to the text "Model results analyzed and interpreted in detail".

At the bottom of the panel, there are buttons for "Add context", "Plan, search, build anything", "Agent", and "Auto". A status bar at the very bottom shows "Ln 1, Col 1 Spaces: 4 UTF-8 LF {} JSON Cursor Tab".

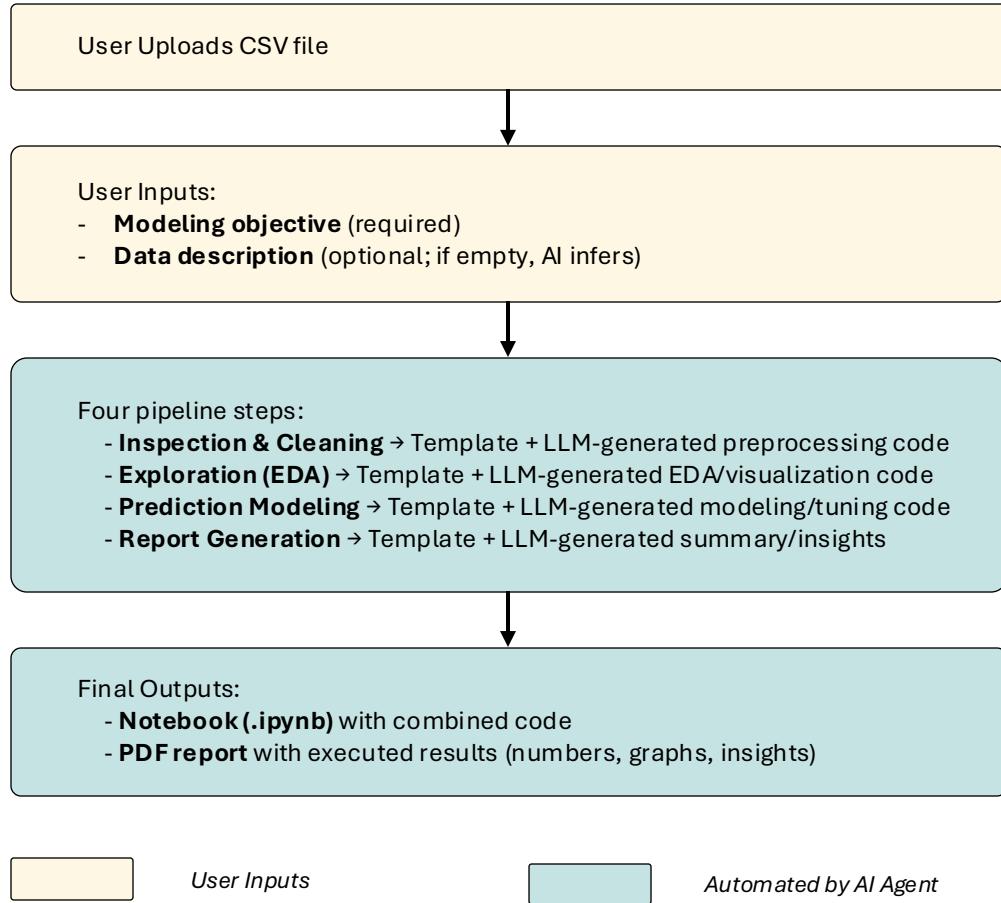
# AGENT STRUCTURE COMPARISON

Comparing Three Approaches for Building an AI-Driven Modeling Pipeline

- Finally, used structure 3 to capture consistency, flexibility, and reliability

STRUCTURE	DESCRIPTION	ADVANTAGES	DISADVANTAGES
<b>1: LLM Fully Generates Code</b>	LLM generates the entire pipeline code (inspection, EDA, modeling, report) from scratch based only on the prompt → saves to notebook → executes it	<ul style="list-style-type: none"><li>- Strong “AI builds everything” impression</li><li>- Highly flexible across datasets and objectives</li><li>- Fully customized code</li></ul>	<ul style="list-style-type: none"><li>- Risk of hitting LLM output length limits</li><li>- Execution/quality failures</li><li>- Difficult to ensure result consistency</li><li>- Code may differ each time</li></ul>
<b>2: Template + LLM Partial Updates</b>	Fixed template governs entire pipeline; LLM only modifies small sections based on data/context (using RAG) → saves to notebook → executes it	<ul style="list-style-type: none"><li>- Always consistent structure</li><li>- High execution success</li><li>- Stable report formats</li><li>- Suitable for real-world/industry use</li></ul>	<ul style="list-style-type: none"><li>- Less “AI builds everything” impact</li><li>- Requires careful template design and maintenance</li></ul>
<b>3: Hybrid (Template + LLM Blocks)</b>	Template provides full pipeline skeleton; LLM generates/enhances core blocks at each step → merges everything into final notebook → executes → generates consistent PDF report	<ul style="list-style-type: none"><li>- Combines template consistency + LLM creativity</li><li>- High execution success - Still gives “AI writes code” feel</li><li>- Most practical and widely used approach</li></ul>	<ul style="list-style-type: none"><li>- Needs logic to merge template and LLM code</li><li>- Slightly higher design/maintenance complexity</li></ul>

# HOW WE IMPLEMENT



- **LangChain Application on Four Pipeline Steps**

PIPELINE STEP	TEMPLATE ROLE	LLM ROLE
Inspection & Cleaning	Data loading, basic structure	Generate custom preprocessing code (missing value handling, outlier removal, type conversion)
Exploration (EDA)	Define EDA cell structure, basic plots	Suggest custom visualizations, correlation analysis, auto-statistics, tailored summaries
Prediction Modeling	Define full modeling pipeline structure	Choose model type, create feature engineering, generate tuning/evaluation code
Report Generation	Provide summary/graph placeholders, report layout	Generate natural-language explanations, insights, business recommendations

## default

POST /analyze Analyze Data

Parameters

No parameters

Request body required

multipart/form-data

modelling\_context \* required  
string  
Your task is to predict 'Gender' based on 'Start Time', 'Trip...c

dataset \* required  
string(\$binary)  
Choose File NYC-BikeShare-2015-2017-combined.csv

dataset\_description \* required  
string  
The dataset contains anonymized trip information for trips m

target\_variable \* required  
string  
'Gender'

independent\_variables \* required  
string  
'Start Time', 'Trip\_duration\_in\_min', 'User Type'

User Inputs Block

Execute Clear

# WHAT NEEDS TO BE ADDED OR MODIFIED

## **Template Code Management**

- Separate files or blocks for each phase (e.g., inspection\_template.py, eda\_template.py)
- Define placeholders:

```
# === LLM_CLEANING_BLOCK_START ===  
# === LLM_CLEANING_BLOCK_END ===
```

## **LLM Code Integration**

- Use LangChain to only generate/edit core logic, not full scripts
- Insert LLM-generated blocks precisely into templates

## **Notebook Assembly & Execution**

- Combine all template + LLM blocks into a unified .ipynb
- Use papermill or similar tools to execute notebook and capture results

## **Consistent Report Generation**

- Define fixed formats for summary, tables, graphs
- Convert executed notebook to PDF using nbconvert for consistent layout

```
# === LLM_INITIAL_SETUP_START ===  
# (LLM will generate code to set up initial parameters and configurations)  
# Example:  
# session_id = '{session_id}'  
# input_file = '{input_file}'  
# output_base_dir = 'eda_mixed_output'  
# === LLM_INITIAL_SETUP_END ===
```

```
"""Custom user/LLM-driven analyses  
== LLM_EDA_CUSTOM_ANALYSIS_START ==  
# (LLM will generate: grouped visual summaries, scatterplots of top correlated pairs, domain-specific insights)  
# (Remove the following examples and replace with LLM-generated code as needed)  
# Example1: User request – Compare the distribution of tripduration by specific group  
if 'usertype' in df.columns and 'tripduration' in df.columns:  
    plt.figure(figsize=(8, 6))  
    sns.boxplot(x='usertype', y='tripduration', data=df)  
    plt.title('Trip Duration by User Type')  
    plt.savefig(f'{output_dir}/tripduration_by_usertype.png')  
    plt.close()  
    log_message("Generated boxplot for tripduration by usertype.", log_file)  
  
# Example2: User request – Visualize the pairs of variables with high correlation  
corr = df.select_dtypes(include=[np.number]).corr()  
high_corr = corr.abs().unstack().sort_values(ascending=False)  
high_corr = high_corr[high_corr < 1].drop_duplicates()  
top_pairs = high_corr.head(3).index  
for var1, var2 in top_pairs:  
    plt.figure()  
    sns.scatterplot(x=df[var1], y=df[var2])  
    plt.title(f'Scatter: {var1} vs {var2}')  
    plt.savefig(f'{output_dir}/scatter_{var1}_vs_{var2}.png')  
    plt.close()  
    log_message(f"Plotted scatter for {var1} vs {var2}.", log_file)  
# === LLM_EDA_CUSTOM_ANALYSIS_END ===  
# (LLM: Remove or replace the above block as needed)  
.....
```

**Enhance the comments to strengthen autonomy of LLM**

```

# --- System prompt for LLM ---
SYSTEM_PROMPT = (
    "You are an expert Python data scientist."
    "Your task is to read the provided code template and, based on the uploaded dataset and user modeling context, "
    "dynamically modify all dataset-dependent parts to generate a final, executable code file."
    "Ensure the code is robust, reproducible, and tailored to the user's dataset and objectives."
    "Only output the final, complete Python code file."
)

def llm_generate_code(template_path, user_inputs, llm_type='openai'):
    llm = get_llm(llm_type)
    template_code = load_code_template(template_path, user_inputs)
    # Compose the prompt: system instruction + template code + context
    prompt_text = SYSTEM_PROMPT + "\n\n" + template_code + "\n\n" + f"# Context: {user_inputs}"
    chain = LLMChain(llm=llm, prompt=PromptTemplate.from_template("{prompt}"))
    code = chain.run({"prompt": prompt_text})
    return code

def inspection_and_cleaning_step(user_inputs, llm_type='openai'):
    template_path = os.path.join('code_template', 'ic.py')
    code = llm_generate_code(template_path, user_inputs, llm_type)
    # Optionally, save or execute the code here
    return code

def eda_step(user_inputs, llm_type='openai'):
    template_path = os.path.join('code_template', 'eda.py')
    code = llm_generate_code(template_path, user_inputs, llm_type)
    return code

def modeling_step(user_inputs, llm_type='openai'):
    template_path = os.path.join('code_template', 'mod.py')
    code = llm_generate_code(template_path, user_inputs, llm_type)
    return code

```

## Integrate code template and Prompt with LangChain

```

def run_agent(user_inputs, data_path, llm_type='openai'):
    ic_code = inspection_and_cleaning_step(user_inputs, llm_type)
    eda_code = eda_step(user_inputs, llm_type)
    mod_code = modeling_step(user_inputs, llm_type)
    report_code = report_step(user_inputs, llm_type)

    session_id = user_inputs['session_id']
    ic_nb = create_step_notebook(ic_code, session_id, 'ic')
    eda_nb = create_step_notebook(eda_code, session_id, 'eda')
    mod_nb = create_step_notebook(mod_code, session_id, 'mod')

```

# OUTPUTS AND LIMITATIONS

## Example Outputs:

Generated Jupyter Notebook (screenshot or example)

Example code snippets from each step (inspection / EDA / modeling)

Prediction summary (tables or graphs, if available)

## Key Limitations:

LLM output length limits → long code generation often gets cut off

LLM-generated code stability issues (execution failures or errors possible)

Requires iterative prompt refinement → costs time and resources

Designing precise prompts for specific analyses is challenging

```
# Load dataset
df = pd.read_csv('./tmp/20250505_015107_882078_NYC-BikeShare-2015-2017-combined.csv')

# Define target variable and independent variables
target_variable = 'Gender'
independent_variables = ['Start Time', 'Trip_duration_in_min', 'User Type']

# 1. Model Performance Summary
print("Model Performance Summary")
print(f"Final Model: Random Forest Classifier")
print(f"Evaluation Score: 0.75")

# 2. Important Variables (e.g., Tree-based model)
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
model.fit(df[independent_variables], df[target_variable])

if 'feature_importances_' in dir(model):
    importance_df = pd.DataFrame({
        'feature': independent_variables,
        'importance': model.feature_importances_
    }).sort_values('importance', ascending=False)
    print(importance_df)
    importance_df.plot.bar(x='feature', y='importance', legend=False)
    plt.title('Feature Importances')
```

**Our agent successfully attempted automated analysis but a pure LLM-driven approach still needs improvements to achieve practical and robust completeness**

## FURTHER STEPS

**In the short term: manage LLM length + enhance post-processing**

**In the mid/long term: automate prompt-template design and adopt customized or fine-tuned models for expanded capabilities**

- **Detailed Plan:**

STAGE	PLAN
Stage 1	Design chunked (stepwise) LLM requests to avoid length limits
Stage 2	Add automatic code validation + repair loops (e.g., post-processing modules to catch errors)
Stage 3	Expand templates: prepare tailored templates for each analysis type (classification, regression, clustering, time series, etc.)
Stage 4	Automate user-specific prompt design (auto-generate tailored prompts by reading dataset features)
Stage 5	Improve models: switch to open-source LLMs or fine-tuned custom models for longer outputs and more specialized analysis

**THANK YOU**

## VALIDATION CHECK ACROSS DOMAINS

- Experiment with **NYC-BikeShare-2015-2017-combined.csv**
- csv file (17 columns x 735503 rows)

Before: Manual

### Time to Insight

Total hours in days

After: Prompt-driven

### Time to Insight

N minutes

- Experiment with **StoreTraffic.csv**
- csv file (36 columns x 181264 rows)

Before: Manual

### Time to Insight

Total 8 hours in 2 days

After: Prompt-driven

### Time to Insight

N minutes

### Code Efficiency

Computing Time

LOC(Length of Codes)

# Urban Mobility Data  
# Public Transportation Data

# Consumer Behavior Data  
# Retail Industrial Data

## FURTHER STEPS

<b>Time to Insight</b>	Measure the total time taken to complete the analysis
<b>Code Efficiency</b>	Compare total lines of code, code reusability, and redundancy (Ex; Manual: 130 LOC vs. AI: 82 LOC (2 redundant blocks removed))
<b>Insight Coverage</b>	Count how many key analysis components were included (Ex; summary stats, correlation, VIF, group analysis)
<b>Human Feedback Quality Control</b>	Evaluate the usefulness and clarity of AI-generated insights from a human perspective
<b>Interaction Efficiency</b>	Measures how effectively the AI produces correct output based on a single prompt without needing multiple follow-ups
<b>Error Rate or Post-correction Count</b>	Tracks errors or manual corrections required after the AI generates the code