UTM
UNIVERSITI TEKNOLOGI MALAYSIA

FACULTY OF COMPUTING
UTM Johor Bahru

SECP3133 HIGH PERFORMANCE DATA PROCESSING

SEMESTER 2 2024/2025

Project Report:

Optimising High-Performance Data Processing for Carlist.my

GROUP	01
SECTION	02
GROUP MEMBERS	1. MARCUS JOEY SAYNER (A22EC0193) 2. MUHAMMAD LUQMAN HAKIM BIN MOHD RIZAUDIN (A22EC0086) 3. CAMILY TANG JIA LEI (A22EC0039) 4. GOH JING YANG (A22EC0052)
LECTURER	DR. ARYATI BINTI BAKRI

TABLE OF CONTENTS

1.0 Introduction

Digital world-roaming societies today rely heavily on data to make their decisions. The modern world requires the ability to store and process data from massive online spaces in order to stay in touch with one's realistic goals and forecasting in data-centric decision-making. The goal of this project is to create and design a high-performance web crawler that will retrieve structured job-related data from Carlist.my.

An important source of information for automotive industry research is Carlist.my, a well-known website that offers real-time listings for new, used, and reconditioned cars in Malaysia. Its broad coverage of cars from private sellers as well as dealerships provides a thorough understanding of the market, which is helpful for research on consumer demand, pricing fluctuations, and market trends. However, several challenges may impact the efficiency of data collection from this site, including slow response times, crawl rate limitations, and the necessity to adhere to ethical scraping practices.

To address these challenges, High Performance Data Processing (HPDP) techniques—such as multithreading and parallel processing—will be applied to enhance system efficiency. The main objective is to optimise the data processing pipeline to achieve greater speed, scalability, and reliability while maintaining responsible and ethical standards in data scraping practices.

1.1 Objectives

To address the challenges of large-scale job data extraction and processing, this project aims to accomplish the following primary goals:

- To create a web crawler that can retrieve at least 100,000 structured records from Carlist.my.
- To clean, process, and store the collected data in a database or other usable formats like CSV or JSON.
- To apply high-performance computing techniques, including multithreading and parallel processing, to optimise crawling and processing speed.

- To conduct a comparative analysis of the system's performance with and without optimisation, focusing on speed, CPU usage, memory usage, and throughput.

1.2 Target Website and Data to be Extracted

Carlist.my is a leading online platform for vehicle listings in Malaysia, offering an extensive database of new, used, and reconditioned vehicles across the country. The platform provides detailed information on various car models, including their specifications, pricing, seller information, and geographical locations. The types of data to be extracted for analysis are summarised in Appendix A (Table A1), which presents an overview of the key car attributes available on Carlist.my.

The data is collected and cleaned for uniformity by removing duplicates and irrelevant entries and stored in a structured format, MongoDB, for analysis and visualisation.

2.0 System Design and Architecture

This section presents the overall system architecture and the collaborative development process used in this project. It outlines the workflow from web scraping to data transformation and evaluation, detailing the architecture involved, the tools and frameworks used by each team member, and the specific roles assigned to ensure an effective division of tasks. The goal is to demonstrate how various technologies and teamwork were integrated to build a scalable and efficient data pipeline.

2.1 Description of Architecture

Figure 1 represents the workflow of a web scraping system followed by big data tools to process, clean, and evaluate scraped data.

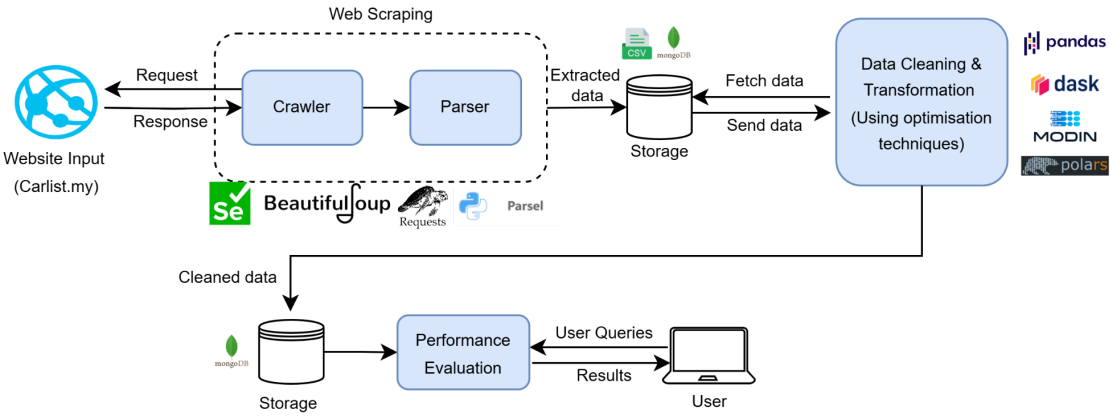


Figure 1: Web Scraping and Big Data Processing Architecture

The components involved in this architecture are described in Appendix B (Table B1), including details on the crawler, parser, data collection process, data transformation, optimisation techniques, and system evaluation metrics.

2.2 Tools and Frameworks Used

In this project, various libraries and frameworks were utilised to efficiently perform web scraping and big data processing tasks. The libraries used by each team member, categorised by their functionality, are summarised in Appendix B (Table B2).

2.3 Roles of Team Members

Each team member was assigned specific roles to ensure efficient division of work and comprehensive coverage of the scraping and data processing tasks, as shown in Table 1.

Table 1 Role Description of Each Team Member

Team Member	**Role**	**Description**
Marcus Joey Sayner	Group Leader and Developer	Coordinated the project and implemented crawling and big data processing using httpx, parsel and Polars.
Muhammad Luqman Hakim bin Mohd Rizaudin	Developer	Implemented crawling and scraping using urllib and Selectolax, and big data processing using Modin.
Camily Tang Jia Lei	Developer	Implemented crawling and scraping using urllib and BeautifulSoup and big data processing using Polars.
Goh Jing Yang	Developer	Implemented crawling and scraping using requests and BeautifulSoup, and big data processing using Dask.

3.0 Data Collection

The data collection process involved scraping used car listings from the Carlist.my website. Each team member independently targeted specific pages on the website and extracted structured information such as car brand, model, manufacture year, mileage, location, price, transmission type, body type, fuel type, and other relevant attributes.

To ensure a wide coverage of data, different page ranges were assigned to each member. After the scraping tasks were completed, the individual datasets from all four members were consolidated into a single comprehensive dataset for further transformation and big data processing.

3.1 Crawling and Scraping Method

The web data extraction process consisted of two primary stages: crawling and scraping. Crawling involved systematically navigating through multiple listings pages on the Carlist.my website to retrieve raw HTML content. Scraping referred to the parsing of this content to extract required data fields such as car name, model, and price. To avoid overloading the website's servers, each crawler was configured with page limits, custom user-agent headers, and delays between requests. The source code for the crawling and scraping implementations is provided in Appendix C (Figures C1 to C3).

3.2 Number of Records Collected

Each team member was tasked with collecting approximately 43,000 records through their individual web scraping efforts. After combining the datasets from all members, a total of around 174,150 car listing records were successfully gathered. This large dataset forms the basis for the subsequent data transformation, optimisation, and analysis tasks.

3.3 Ethical Considerations

Throughout the data collection process, ethical web scraping practices were observed:

- **Polite Scraping:** A user-agent header was used to mimic a real browser, and appropriate delays were implemented between requests to avoid overloading the server.
- **No Data Misuse:** Only publicly available data was collected. Any sensitive information, user personal details, or any hidden information was not targeted.
- **Transparency and Attribution:** The source of the data (Carlist.my) is acknowledged in the report, respecting intellectual property rights.

4.0 Data Processing

Data processing is considered a critical phase in any data-driven project, during which raw data is cleaned, transformed, and structured to meet the requirements of subsequent analysis. The reliability of the final results is shaped by the quality and consistency of the data, both of which are ensured during this stage. In this project, four libraries—Pandas, Polars, Dask, and Modin—were employed for data processing. Each library is recognised for its distinctive strengths in data handling and manipulation, offering scalable and high-performance solutions suited to datasets of varying sizes.

4.1 Cleaning Methods

Data cleaning is regarded as a critical step in ensuring the consistency, accuracy, and usability of a dataset. During the preprocessing stage, various operations were undertaken to resolve redundancies, address missing values, and standardise data formats. As presented in Appendix D (Table D1), key cleaning techniques were applied, including column merging, string manipulation, and the treatment of null entries, in order to prepare the dataset for subsequent analysis.

4.2 Data Structure (CSV/JSON/database)

In this project, the capabilities of four robust data processing libraries—Pandas, Polars, Modin, and Dask—were demonstrated in managing structured data from CSV and JSON files. The datasets were cleaned, transformed, and efficiently exported to a MongoDB database. Each library was selected for its optimisation features in data manipulation and processing, with distinct strengths observed:

- **Pandas** was recognised for its simplicity and flexibility in handling small to medium-sized datasets.
- **Polars**, implemented in Rust, was optimised for high performance and memory efficiency.
- **Modin** offered scalability by enabling parallel processing through Dask or Ray with minimal code changes.

- **Dask** supported adaptive parallel computation, allowing processing of datasets that exceeded memory capacity by dividing tasks into smaller partitions.

To maintain consistency, data in both CSV and JSON formats were processed and stored in MongoDB using a uniform collection schema across all libraries. This uniformity allows for a reliable comparison of performance and resource utilisation across different execution environments.

The entire process reflects the practical applicability of each library within real-world ETL (Extract, Transform, Load) pipelines, facilitating the identification of the most efficient tool for varying data workloads. Figure 2 presents the list of cleaned datasets stored in the MongoDB database.

Figure 2: Cleaned Datasets Stored in MongoDB Database

The cleaned datasets from Pandas, Polars, Modin, and Dask were all processed through similar data cleaning workflows: handling missing values, removing irrelevant records, and reordering fields. Each dataset was then transformed into tabular and JSON formats and stored as collections in MongoDB for efficient querying. Figures 3 to 10 display the JSON and tabular format outputs generated using four data processing libraries: Pandas, Polars, Modin, and Dask.

Pandas (listings_pandas_cleaned):

Pandas was used for its straightforward syntax and strong community support. Figures 3 and 4 show the cleaned data in tabular and JSON formats, respectively, stored in MongoDB.

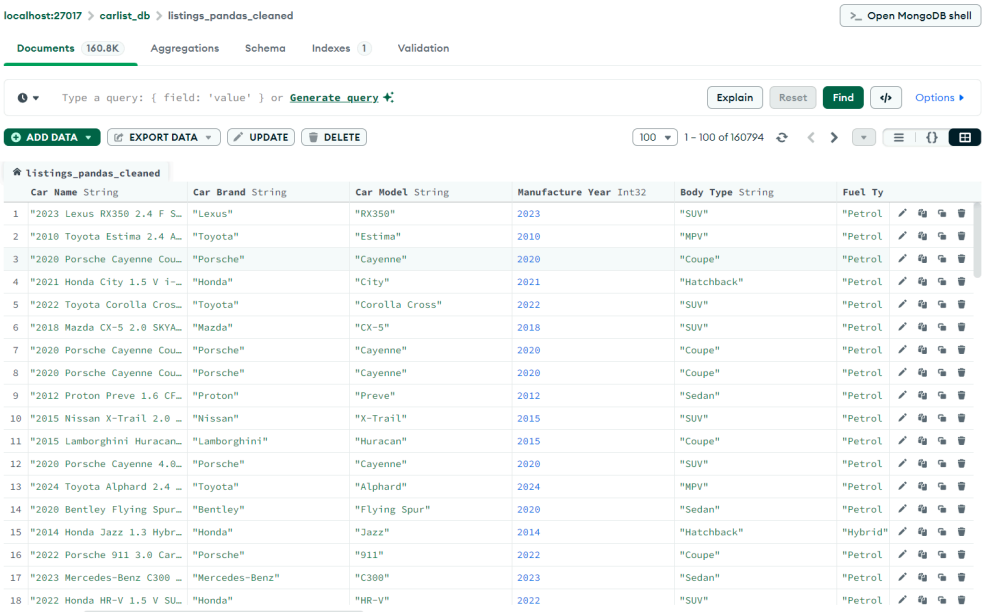


Figure 3: Table-Format View of Pandas-Processed Dataset in MongoDB

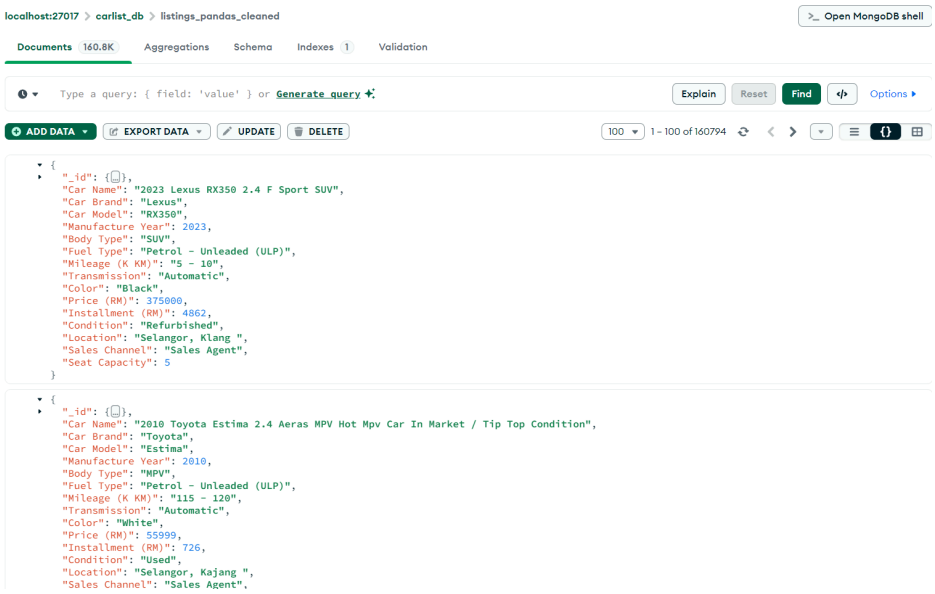


Figure 4: JSON Output of Pandas-Processed Dataset in MongoDB

Polars (listings_polars_cleaned) :

Polars, written in Rust, offered noticeably faster execution and lower memory usage during transformation. See Figures 5 and 6 for its output formats.

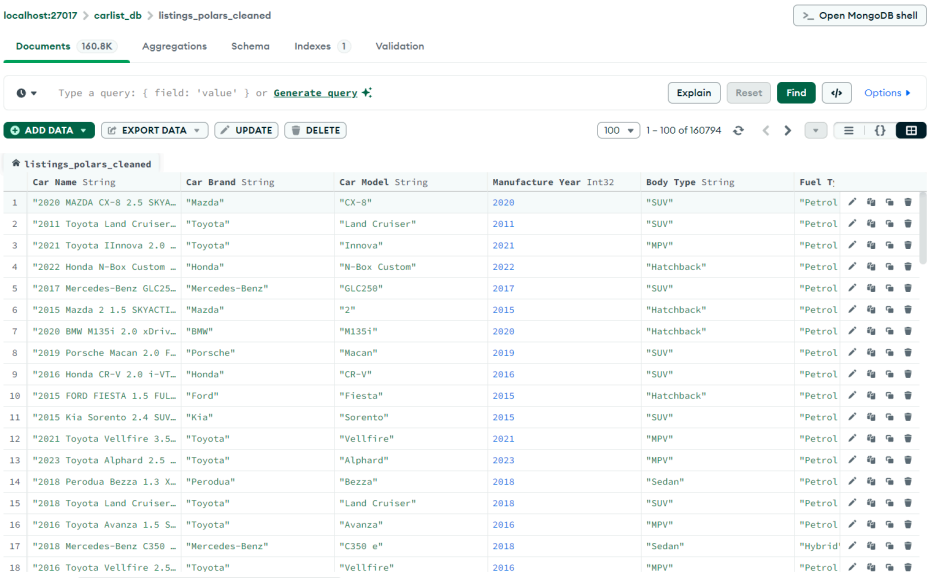


Figure 5: Table-Format View of Polars-Processed Dataset in MongoDB

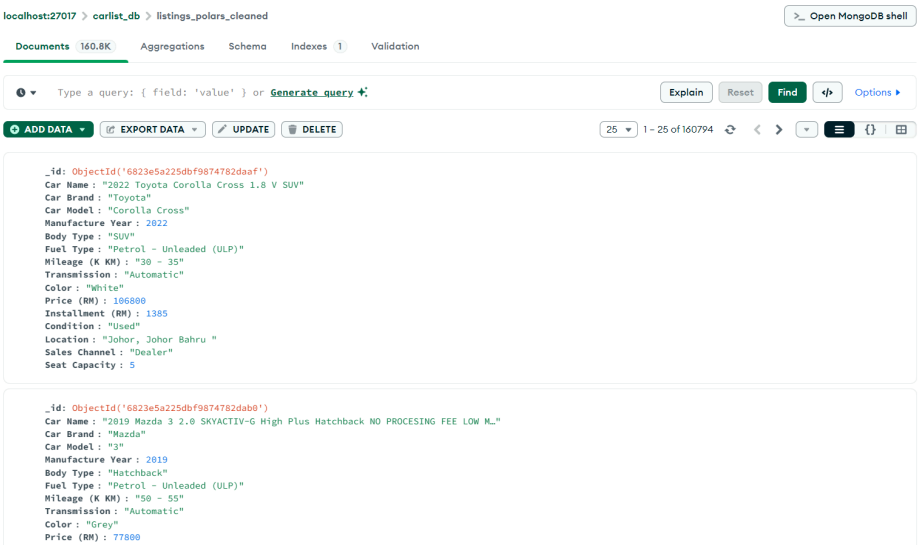


Figure 6: JSON Output of Polars-Processed Dataset in MongoDB

Modin (listings_modin_cleaned):

Modin enabled parallel processing with minimal code changes from Pandas, benefiting from scalability. Figures 7 and 8 show its processed outputs.

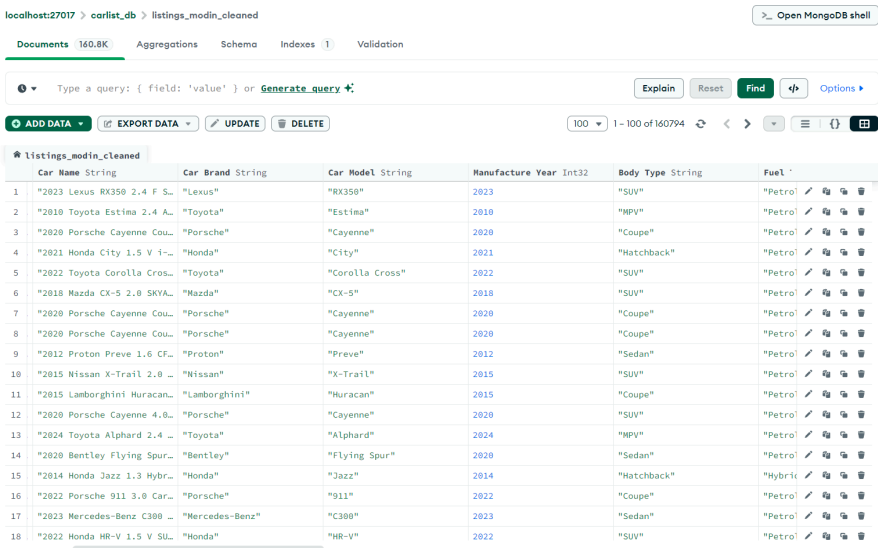


Figure 7: Table-Format View of Modin-Processed Dataset in MongoDB

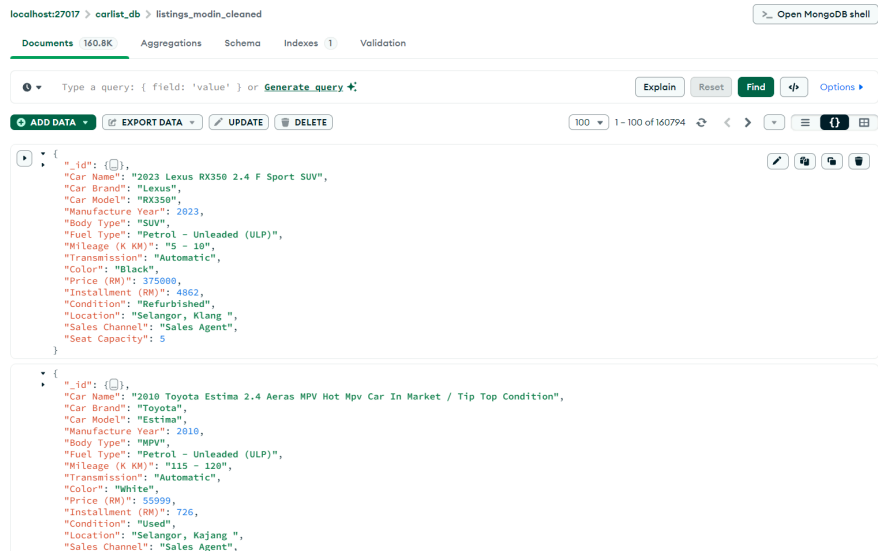


Figure 8: JSON Output of Modin-Processed Dataset in MongoDB

Dask (listings_dask_cleaned):

Dask efficiently handled large datasets by processing them in partitions. Figures 9 and 10 show its outputs stored in MongoDB.

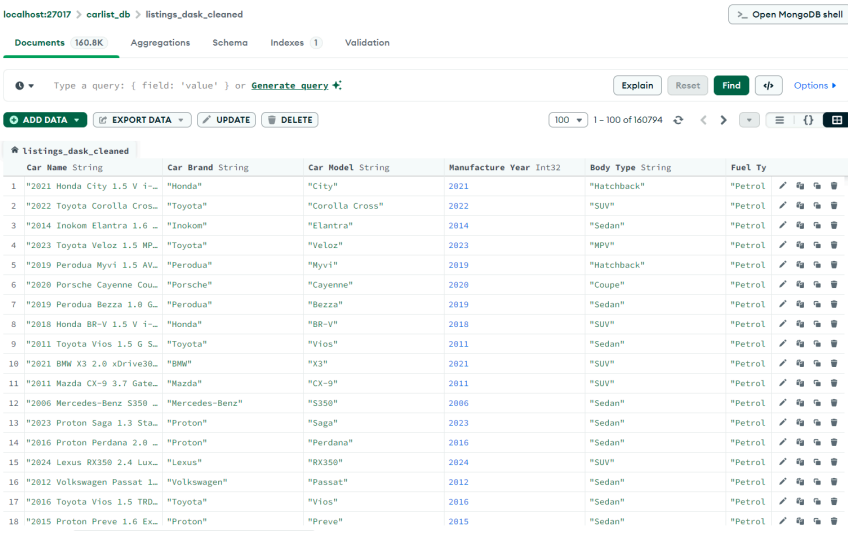


Figure 9: Table-Format View of Dask-Processed Dataset in MongoDB

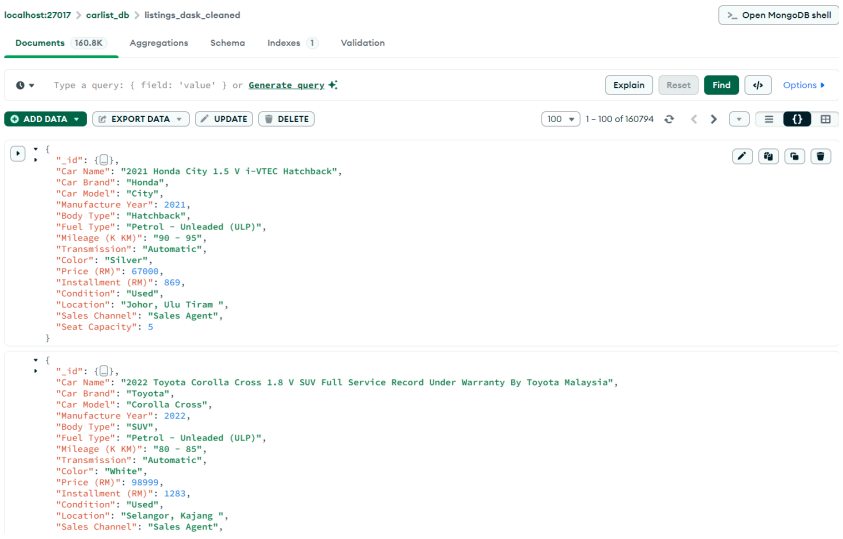


Figure 10: JSON Output of Dask-Processed Dataset in MongoDB

4.3 Transformation and Formatting

4.3.1 Combining Redundant Columns

Target Columns: 'Seat Capacity' and 'Seating Capacity'

Objective: To eliminate semantic redundancy by merging similar columns.

Process: In instances where the ‘Seat Capacity’ column was null but ‘Seating Capacity’ contained a value, the value from ‘Seating Capacity’ was used to fill in the missing data. If ‘Seat Capacity’ already had a value, it was retained. After the merging process, the ‘Seating Capacity’ column was dropped to avoid redundancy. Figure 11 illustrates the Pandas code implementation used to perform this column combination and removal.

```
# === Data Cleaning Step: Combine 'Seat Capacity' and 'Seating Capacity' ===
total_rows_combinecol_pandas = df_pandas.shape[0]

# Combine 'Seat Capacity' and 'Seating Capacity' into one column
df_pandas['Seat Capacity'] = df_pandas['Seat Capacity'].combine_first(df_pandas['Seating Capacity'])

# Drop 'Seating Capacity' column
df_pandas.drop(columns=['Seating Capacity'], inplace=True)
```

Figure 11: Pandas Code for Combining Redundant Columns: 'Seat Capacity' and 'Seating Capacity'

4.3.2 Removing Noisy and Irrelevant Columns

Target Column: 'URL'

Objective: To eliminate noisy and irrelevant data.

Method: The ‘URL’ column was identified as analytically irrelevant and subsequently removed to reduce dataset dimensionality and eliminate noise. This operation is illustrated in Figure 12.

```
# === Data Cleaning Step: Remove 'URL' column ===
total_rows_removecol_pandas = df_pandas.shape[0]

# ================================================

# Remove the 'URL' column
df_pandas.drop(columns=['URL'], inplace=True)

# ================================================
```

Figure 12: Pandas Code for Removing Noisy Column: 'URL'

4.3.3 String Cleaning and Formatting

This section details the preprocessing steps applied to key textual columns to ensure consistency and enable numerical analysis. It covers cleaning monetary strings in the Installment column, extracting categorical labels from schema URLs in the Condition column, standardising sales source identifiers in the Sales Channel column, and binning mileage values into defined intervals. Code examples demonstrating these transformations are provided in Figures 13 through 16.

4.3.3.1 Installment Column

Objective: To clean monetary values for numeric operations.

Method: Monetary string values were cleaned by removing the suffix '/month', the prefix 'RM', and any commas. The cleaned strings were then converted to numeric format to enable mathematical operations. The code for this conversion is presented in Figure 13.

```
# === Data Cleaning: Clean 'Installment' Column ===
total_rows_cleaninstallment_pandas = df_pandas.shape[0]

# Cleaning operations
df_pandas['Installment'] = df_pandas['Installment'].str.replace('RM', '', regex=False)
df_pandas['Installment'] = df_pandas['Installment'].str.replace(',', '', regex=False)
df_pandas['Installment'] = df_pandas['Installment'].str.replace('/month', '', regex=False)
df_pandas['Installment'] = pd.to_numeric(df_pandas['Installment'], errors='coerce')
```

Figure 13: Pandas Code for Cleaning the 'Installment' Column

4.3.3.2 Condition Column

Objective: To extract simple categorical labels from schema URLs.

Method: Schema URLs in the Condition column were parsed to extract simplified categorical labels such as 'Used' or 'New' (e.g., from https://schema.org/UsedCondition). This transformation of schema-based condition values into human-readable categories is reflected in Figure 14.

```
# === Data Cleaning: Extract Clean Values from 'Condition' Column ===
total_rows_condition_pandas = df_pandas.shape[0]

# Extract value from '...schema.org/XCondition'
df_pandas['Condition'] = df_pandas['Condition'].apply(
    lambda x: re.search(r'\.org/([A-Za-z]+)Condition', x).group(1) if isinstance(x, str) and 'schema.org/' in x else x
)

# Strip 'Condition' if it remains (for post-processed cases)
df_pandas['Condition'] = df_pandas['Condition'].str.replace(r'Condition$', '', regex=True)
```

Figure 14: Pandas Code for Extracting Categorical Values from the 'Condition' Column

4.3.3.3 Sales Channel Column

Objective: To simplify and standardise sales source identifiers.

Method: Textual content in the *Sales Channel* column was parsed and categorised using regular expressions to standardise valid entries such as 'Sales Agent' or 'Dealer'. This transformation process is demonstrated in Figure 15.

```
# === Data Cleaning: Clean 'Sales Channel' Column ===
total_rows_saleschannel_pandas = df_pandas.shape[0]

# Extract Sales Channel (either 'Sales Agent' or 'Dealer')
df_pandas['Sales Channel'] = df_pandas['Sales Channel'].str.extract(r'^(Sales Agent|Dealer)')
```

Figure 15: Pandas Code for Standardising the 'Sales Channel' Column

4.3.3.4 Mileage Column

Objective: To standardise mileage ranges into bins.

Method: Suffixes such as *'K KM'* were removed, and mileage ranges (e.g., '5 - 10') were parsed. Single values were binned into predefined intervals (e.g., 12 was binned into '10 15'). The binning logic is presented in Figure 16.

```python
# === Data Cleaning: Clean 'Mileage' Column ===
total_rows_mileage_pandas = df_pandas.shape[0]

# Remove "K KM" and clean mileage into 5K step ranges
def process_mileage_pandas(mileage):
    if not isinstance(mileage, str) or mileage.strip() == '':
        return None
    try:
        if '-' in mileage:
            start, end = mileage.split('-')
            return f"{int(start.strip())} - {int(end.strip())}"
        else:
            val = int(mileage.strip())
            lower = (val // 5) * 5
            upper = lower + 5
            return f"{lower} - {upper}"
    except:
        return None

df_pandas['Mileage'] = df_pandas['Mileage'].str.replace('K KM', '', regex=False)
df_pandas['Mileage'] = df_pandas['Mileage'].apply(process_mileage_pandas)
```

Figure 16: Pandas Code for Binning Values in the 'Mileage' Column

4.3.4 Handling Missing Data

Objective: To retain only rows with complete critical data.

Critical Columns Checked: 'Body Type', 'Fuel Type', 'Mileage', 'Sales Channel', and 'Seat Capacity'

Method: Rows containing null values in any of the critical columns were removed using the dropna() function. This approach was taken to preserve data integrity without introducing imputation bias. The implementation is detailed in Figure 17.

```python
# ============================================================================

# === Data Cleaning Step: Drop rows with missing values ===
total_rows_before_cleaning_null_pandas = df_pandas.shape[0]

# Drop rows with missing values in specific columns
df_pandas.dropna(subset=['Body Type', 'Fuel Type', 'Mileage', 'Sales Channel', 'Seat Capacity'], inplace=True)

# ============================================================================
```

Figure 17: Pandas Code for Handling Missing Data in Critical Columns

4.3.5 Renaming Columns for Clarity

Objective: To add units to numeric columns for readability.

Renaming Map:

'Mileage' → 'Mileage (K KM)'

'Price' → 'Price (RM)'

'Installment' → 'Installment (RM)'

Method: Figure 18 shows rename columns of the dataset by appending units to avoid ambiguity: For instance, Price to Price (RM), and Mileage to Mileage (K KM). This kind of detailing unquestionably refines the understanding of the data by specifying the units of measurement.

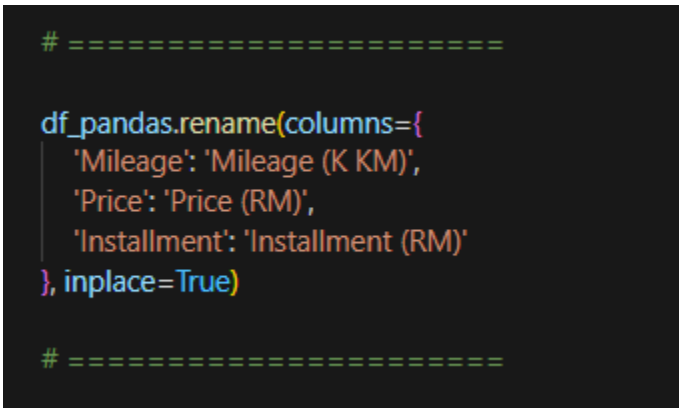


Figure 18: Pandas Code for Renaming Columns for Clarity

4.3.6 Removing Duplicate Rows

Objective: To prevent duplicate entries from distorting analysis.

Method: Duplicate entries were identified using .duplicated() and removed using .drop_duplicates(), ensuring that exact duplicates were eliminated while retaining valid lookalikes. The applied logic is captured in Figure 19.

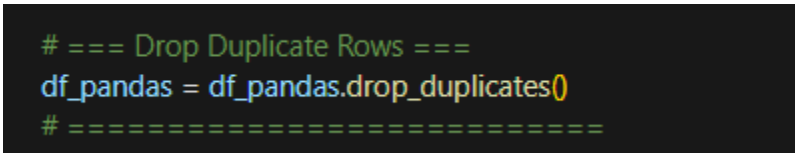


Figure 19: Pandas Code for Removing Duplicate Rows

5.0 Optimisation Techniques

In data preprocessing, optimisation techniques are considered crucial for efficiently handling large-scale datasets. The performance of a data pipeline can be improved by selecting libraries that offer optimisation capabilities such as parallel computing and multi-threading. In this project, the performance of four Python-based data processing libraries—Pandas, Polars, Modin, and Dask—is investigated by applying equivalent cleaning and transformation operations to a combined car listings dataset. Each library is evaluated based on execution time (wall time), CPU usage and CPU time, memory usage (current and peak), and processing throughput (rows per second). Through this evaluation, the objective is to determine how different libraries optimise resource usage and processing time when identical data cleaning operations—such as column combination, string manipulation, and handling of missing values—are performed.

5.1 Methods Used

To ensure a fair comparison, identical data cleaning steps were applied across all four libraries. The performance optimisation techniques specific to each library are detailed in Appendix E (Table E1).

5.2 Code Overview

This section presents the code implementations used to apply optimisation techniques in each data processing library. While the data cleaning and transformation steps were standardised across all libraries, the code was written separately to accommodate the unique syntax and execution model of each library. The content is organised into three parts: the task performed, the library used, and the corresponding code snippet. Screenshots are included for reference, and a summary of all tasks and associated libraries is provided in Appendix E (Table E2).

6.0 Performance Evaluation

Performance evaluation is conducted to assess the effectiveness of the optimisation techniques applied in this project. This evaluation focuses specifically on the data processing component by comparing how different libraries perform identical data cleaning tasks under various optimisation methods. The objectives of the performance evaluation are to achieve:

- Reduced processing time
- Lower memory consumption
- Improved CPU utilisation
- Higher throughput (i.e., more records processed per minute)

6.1 Before vs After Optimisation

This section presents a comparison of the performance before and after applying optimisation techniques using different libraries: Pandas, Polars, Modin, and Dask.

- Pandas is used as the baseline, representing a non-optimised, single-threaded processing approach.
- In contrast, Polars, Modin, and Dask represent optimised solutions.
    - Polars utilises multi-threaded execution and is implemented in Rust to achieve high performance.
    - Modin introduces parallel execution by distributing tasks across multiple CPU cores using backend engines such as Dask.
    - Dask employs task-based parallelism with lazy evaluation, enabling it to efficiently scale data processing workflows.

By comparing the results obtained from performing the same data cleaning tasks, the impact of these optimisation techniques on overall system performance can be assessed.

6.2 Time, Memory, CPU Usage, Throughput

To evaluate performance, six metrics were collected during data processing tasks using each library. These metrics were consistently measured using the Python libraries time, psutil, and tracemalloc. The metrics include:

- **Elapsed Time (Wall Time) (s):** The real-world time taken to complete the task from start to finish.
- **CPU Time (ms):** The actual CPU processing time consumed, measured in milliseconds.
- **CPU Usage (%):** The intensity of CPU usage during code execution.
- **Throughput (rows/sec):** The number of records processed per second, calculated as total rows divided by elapsed time.
- **Current Memory Usage (MB):** The amount of memory in use at the end of execution.
- **Peak Memory Usage (MB):** The highest amount of memory used at any point during execution.

The compiled results for these metrics across all libraries are summarised in Appendix F (Table F1).

6.3 Charts and Graphs

Figure 20 presents a comparison of four Python data-processing libraries—Pandas, Polars, Modin, and Dask—across a typical ETL pipeline. Polars is observed to perform the fastest overall, particularly in the parsing, bucketing, and MongoDB export stages. Pandas demonstrates the quickest performance in data import and simple column operations but shows moderate efficiency on more complex string-cleaning tasks. Higher overheads are incurred by Modin and Dask, especially during import, null-filling, and memory optimisation steps, rendering them more beneficial primarily at larger scales or within distributed environments. Consequently, Polars is identified as the preferred choice for accelerating most in-memory workflows, whereas Pandas remains advantageous for straightforward, single-machine operations.

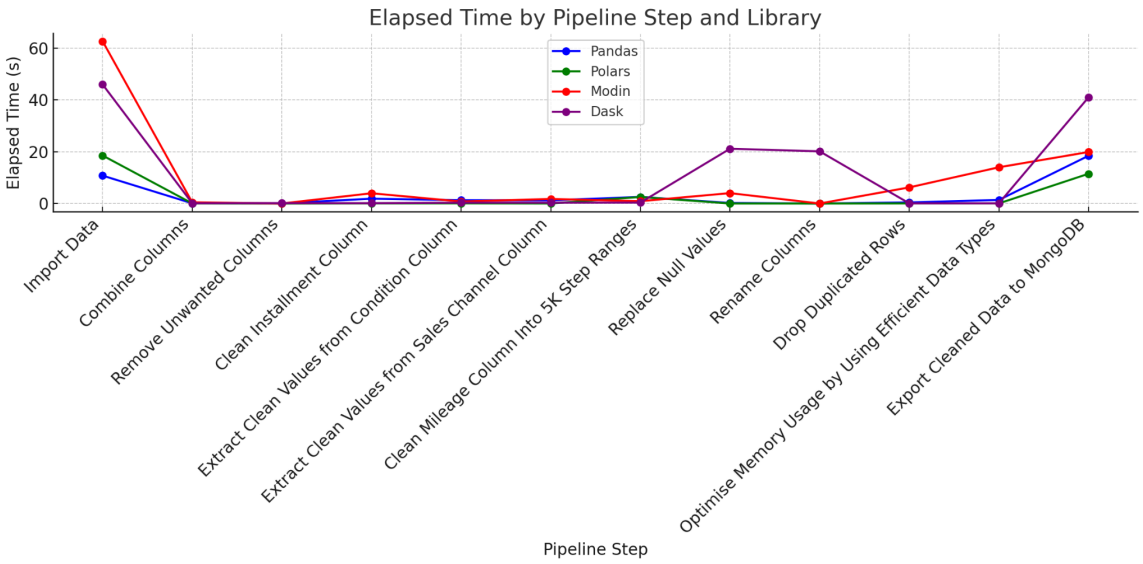


Figure 20: Line Graph of Elapsed Time by Pipeline Step and Library

Figure 21 shows that Polars is by far the most CPU-efficient, using the fewest milliseconds for virtually every ETL step, while Pandas is positioned in the middle, with low overhead on simple operations but increasing on heavier parsing and export. Huge CPU cycles are consumed by Modin on import (~180,000 ms) and memory tuning, after which the usage levels off to a similar but still higher amount than Pandas on string-cleaning and MongoDB writes. Significant CPU overhead is also observed in Dask on broadcast-style tasks (null-replacement, memory optimisation) and export, which reflects its chunked and distributed scheduling costs. In summary, the leanest CPU footprint is delivered by Polars, Pandas is considered reasonable for small-scale in-memory work, and Modin and Dask only prove advantageous when distributed throughput is required.

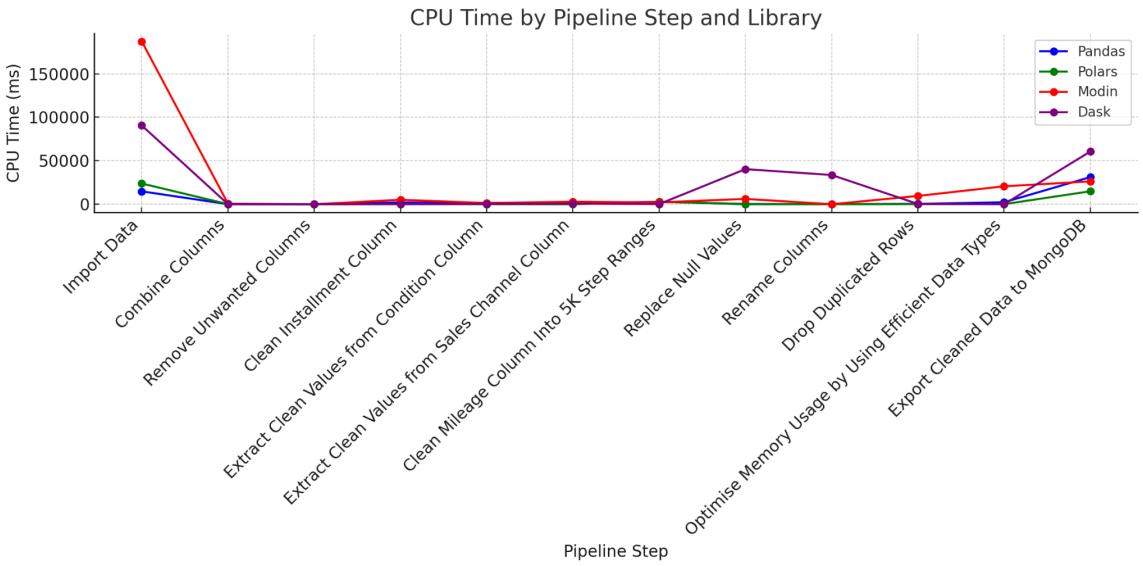


Figure 21: Line Graph of CPU Time by Pipeline Step and Library

Figure 22 demonstrates that Polars consistently maintains the smallest and steadiest CPU footprint, with utilisation pegged between approximately 6% and 16% throughout the pipeline, which is attributed to its efficient, multi-threaded Rust core. Pandas is positioned in the middle, consuming about 7%–20% of CPU with no dramatic spikes, reflecting its straightforward, single-threaded execution. By contrast, utilisation is driven above 25% by Modin's parallel scheduler on simple combines and approaches 45% when dropping duplicates (with 12%–20% elsewhere), while Dask likewise increases to around 20% for broadcast-style tasks such as null replacement and memory down-casting (and otherwise remains near 8%–15%). In practice, Polars is identified as the most CPU-efficient choice, Pandas offers stable, moderate usage, and Modin and Dask justify their additional overhead only when parallel or distributed throughput is truly required.

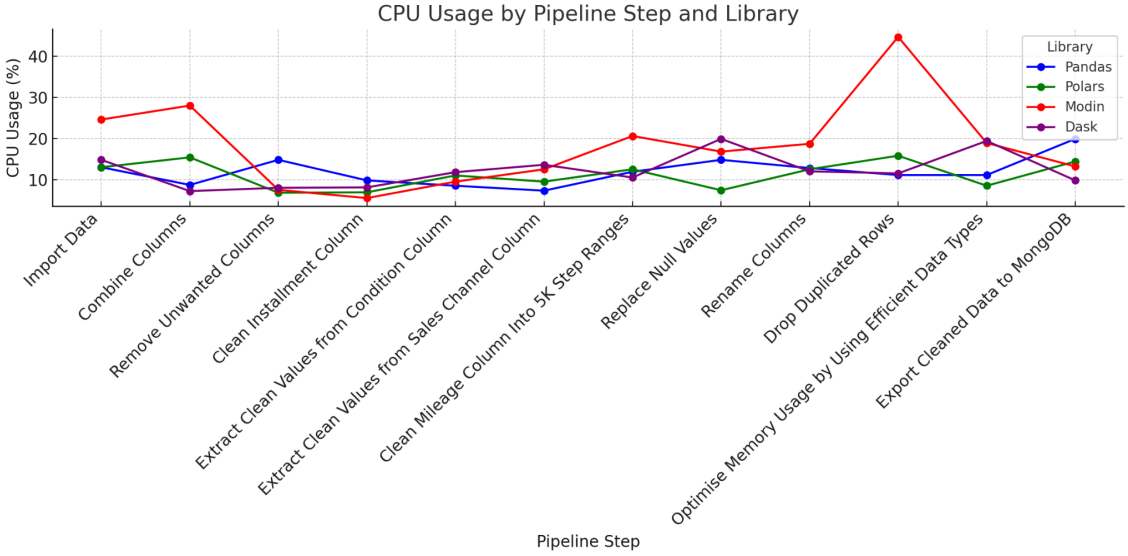


Figure 22: Line Graph of CPU Usage by Pipeline Step and Library

Figure 23 illustrates the number of rows processed per second by each library at various ETL steps. Polars significantly outperforms the others, achieving approximately 150 million rows per second during string-cleaning and renaming tasks and maintaining 80–90 million rows per second during bucketing and channel-parsing. Pandas ranks second, peaking at around 120 million rows per second during the efficient dtype-conversion step and processing 30–50 million rows per second for most cleaning operations. Moderate performance improvements are observed with Modin's parallel engine, which processes approximately 30–40 million rows per second on intensive parsing and renaming tasks, though it does not reach the performance levels of Polars. Dask, when executed on a single node, exhibits significantly lower throughput (generally under 1 million rows per second) due to the overhead imposed by its chunked scheduling approach on these in-memory workloads. In summary, Polars demonstrates the highest throughput, Pandas performs respectably, Modin provides some additional speed, and Dask proves advantageous primarily when scaled across multiple machines.

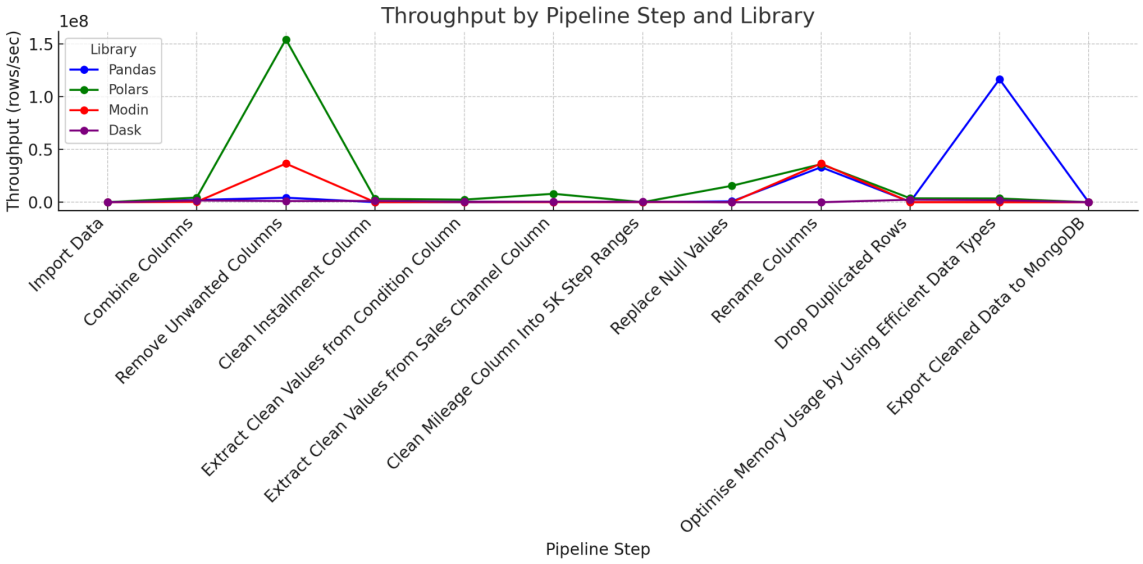


Figure 23: Line Graph of Throughput by Pipeline Step and Library

Figure 24 presents the current memory usage by pipeline step for each library. After loading the CSV file, all four libraries occupy approximately 420–450 MB of RAM; however, their steady-state memory footprints diverge markedly during subsequent transformations. Polars promptly releases its buffers and maintains an additional resident memory usage close to 0 MB, consistent with its streaming architecture. Pandas sustains approximately 20–30 MB of memory usage throughout the cleaning steps, whereas Modin generally holds a slightly lower footprint of around 10–15 MB during most string parsing and renaming operations, increasing to roughly 15 MB during export. Dask's memory usage fluctuates, alternating between near-zero usage on simple column operations and a per-chunk overhead, ranging from approximately 5 MB during null replacement to up to 25 MB during the final MongoDB write. Overall, Polars demonstrates the most minimal working set, Pandas and Modin maintain moderate memory buffers, and Dask's footprint varies in accordance with its chunked execution model.

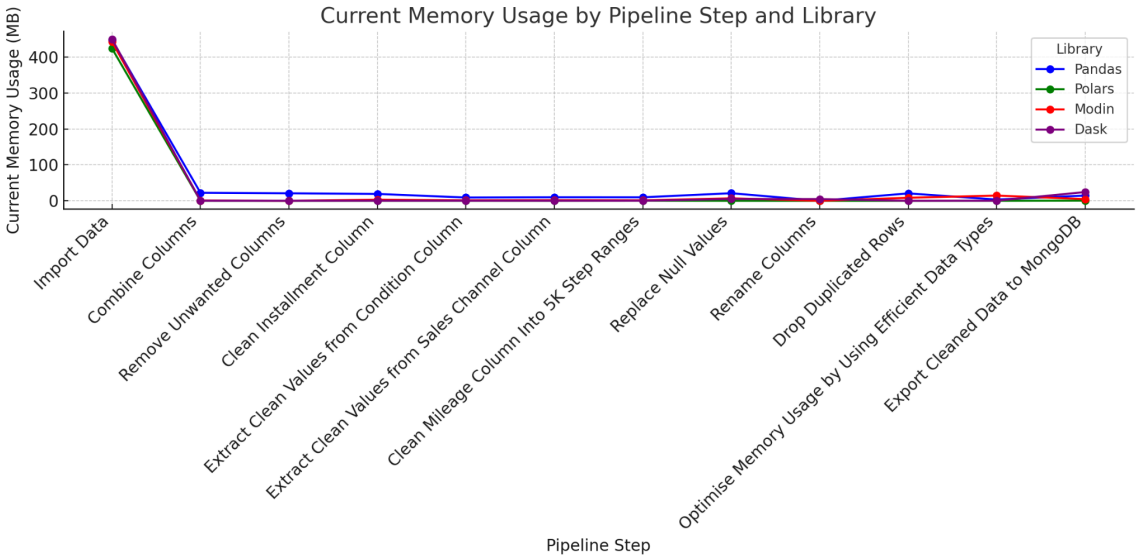


Figure 24: Line Graph of Current Memory Usage by Pipeline Step and Library

Figure 25 illustrates the peak memory usage by pipeline step for each library. It is evident that reading the CSV file constitutes the largest single memory allocation, with Dask exhibiting a peak of approximately 800 MB, Modin 550 MB, Pandas 520 MB, and Polars 430 MB. Following this initial load, Polars demonstrates the smallest additional memory peaks, ranging from single-digit to low-teens MB during string parsing and bucketing operations. In contrast, Pandas and Modin reach peak usages between 15 and 35 MB throughout their various cleaning steps. Dask, however, maintains a substantial peak of 340 MB during broadcast-style operations such as null replacement and memory down-casting, before decreasing to near zero during simpler column operations. During the MongoDB export phase, notable transient peaks are again observed, with Dask and Polars reaching approximately 340 MB and 270 MB, respectively, while Modin and Pandas peak at 200 MB and 180 MB. All in all, Polars minimises additional memory consumption following the initial load, Pandas and Modin incur moderate buffer peaks, and Dask exhibits the highest transient overhead within this single-node ETL workflow.

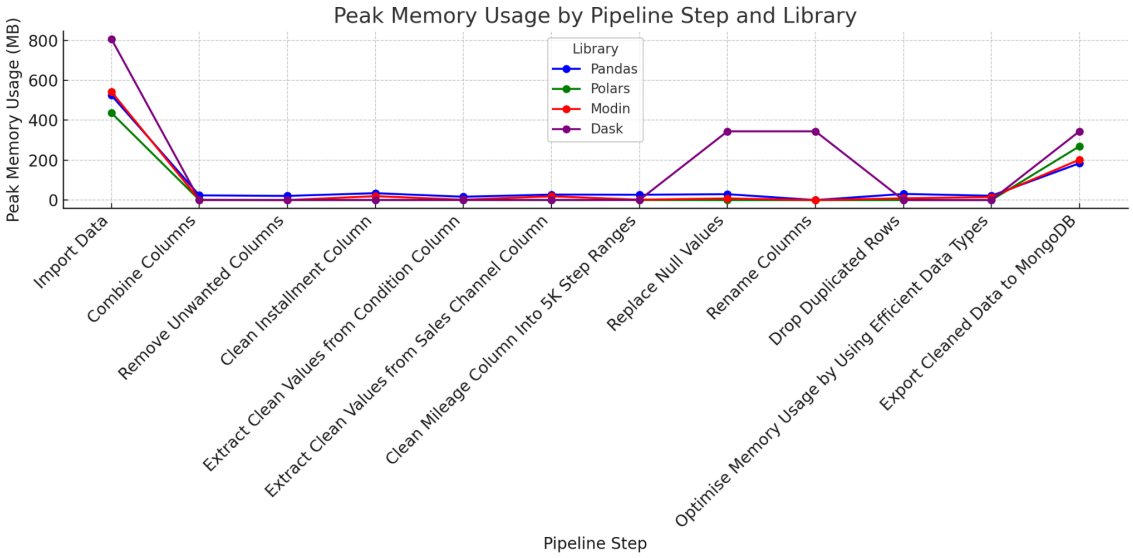


Figure 25: Line Graph of Peak Memory Usage by Pipeline Step and Library

7.0 Challenges and Limitations

Several difficulties were encountered when scraping automobile listings using Visual Studio Code (VSCode). One major issue was that the script would unexpectedly stop, causing VSCode to freeze—especially during attempts to scrape large volumes of data from the website. This inability to complete the task efficiently significantly hampered both progress and overall productivity.

In addition, web scraping was limited by restrictions in robots.txt and the availability of accessible records. The robots.txt file may block certain paths, rendering it illegal or unethical to scrape specific types of content. Furthermore, some websites implement poor pagination or display only a limited number of items per page, making it difficult to collect large datasets effectively.

Another challenge stemmed from the incompatibility of some Python libraries with the Google Colab environment. Libraries such as Playwright and Selenium encountered installation or runtime issues due to Colab's limited support for headless browsers and restricted system-level operations. Additionally, Colab’s memory and session time limits imposed further constraints when processing large volumes of scraped data.

7.1 Reasons of Challenges and Limitations

The freezing issue in VSCode was likely caused by the high resource demands of running the scraping script directly within the IDE. The integrated terminal in VSCode struggled to handle the large amount of data, especially when real-time output was printed extensively. As a result, VSCode became unresponsive during execution.

The restriction imposed by robots.txt exists to manage web traffic, protect user data, and prevent server overload. Website owners use this file to define which bots may access their content. Limited accessible records may also result from dynamic content loading via JavaScript, API constraints, or intentional design choices aimed at discouraging scraping.

Google Colab, although effective for quick data analysis and learning machine learning models, does not perform well when high computing power or large-scale data processing is required. Its lack of support for running long background jobs and modifying system-wide settings limits the capabilities of tools that require persistent browser control. In addition, Colab's RAM is typically restricted to 12–16 GB, and execution time is limited, which complicates long-running scraping and data-gathering tasks.

7.2 Solutions

To address the freezing issue encountered in Visual Studio Code (VSCode), several adjustments were implemented. The scraping process was divided into two smaller tasks. Each request was delayed by approximately two to five seconds to reduce the workload on both the target server and the local system. Data was incrementally saved between each batch to manage memory usage and prevent overload. The resulting CSV files from each segment were subsequently merged to create a single comprehensive dataset. These modifications effectively mitigated system failures and improved overall performance.

Ethical scraping practices were observed by adhering to the website's usage policies. In instances where direct access to resources was restricted, publicly available alternatives and open APIs were utilised. Automation tools such as Playwright and Selenium were employed to emulate user interactions and load dynamically generated content, even when data availability was limited. To avoid detection or triggering of anti-scraping mechanisms, data collection was evenly distributed across multiple categories and pages, with deliberate time delays between requests.

To overcome the limitations of Google Colab, the httpx library was combined with lightweight parsers such as Selectolax or BeautifulSoup. This approach reduced dependency overhead and offered more stable performance within Colab’s constrained environment. For large-scale data processing, local servers or cloud services such as Google Cloud Platform, Amazon Web Services (AWS), or Microsoft Azure were recommended. Intermediate results were stored using platforms like Google Drive or BigQuery, and the scraping tasks were divided into smaller batches to optimise resource allocation.

8.0 Conclusion and Future Work

This section presents key findings derived from the data collection and processing activities conducted on various websites. It evaluates the performance of each library and outlines the major challenges encountered during their use. Besides, it offers recommendations and practical suggestions to improve the efficiency, reliability, and scalability of future implementations.

8.1 Summary of Findings

From an ETL pipeline perspective, Polars demonstrates superior performance in terms of speed, CPU efficiency, and memory usage compared to Pandas, Modin, and Dask. This makes it a highly suitable option for in-memory processing of large datasets. In terms of overall performance, Polars consistently outperforms the other libraries, particularly in tasks involving data parsing, transformation, and migration to MongoDB. It recorded the lowest CPU usage (6%–16%) and the smallest memory footprint throughout the processing stages. These advantages are largely attributed to its Rust-based backend, which is optimised for multi-threading and capable of handling up to 150 million rows per second.

Although slower in handling intensive operations, Pandas remains effective for simpler tasks. It maintains a low to moderate CPU load (7%–20%) and exhibits stable memory consumption across most ETL stages. For small-scale workflows executed on a single machine, Pandas continues to serve as a reliable and widely adopted solution.

Modin and Dask, which are designed for distributed computing across multiple machines, incur higher initialisation overhead, particularly during data import and missing value imputation. Modin tends to consume higher CPU resources during large-scale operations, yielding performance gains primarily when true parallelism is required. Dask, due to its distributed task scheduler, demonstrates suboptimal performance on single-machine setups, with relatively low throughput and notable memory fluctuations during specific operations.

Upon initial CSV loading, all libraries register similar memory usage in the range of 420–450 MB. However, their memory behaviour diverges during processing. Polars, benefiting from a streaming architecture, exhibits the most efficient memory handling. In contrast, Pandas and Modin manage data using smaller buffers, while Dask shows the highest variability—peaking at 800 MB during loading and settling at approximately 340 MB during transformations.

In conclusion, for a standalone ETL workflow, Polars offers the best overall performance in terms of processing speed, resource efficiency, and scalability when compared to Pandas, Modin, and Dask.

8.2 Improvement for Future Works

Several improvements should be implemented to ensure that future web scraping and data processing can be conducted reliably and at scale. Firstly, establishing a stable and flexible computing environment is essential. It is recommended to use local machines equipped with sufficient resources or cloud services such as AWS EC2, Azure, or DigitalOcean, as they support a broader range of tools and offer greater flexibility. Tools such as Playwright and Selenium are better suited for these environments, since platforms like Google Colab do not fully support headless browsers and impose runtime limitations (Mitchell, 2021; Amazon Web Services, 2023).

Robust error-handling mechanisms are also critical to maintaining continuous operation during intensive scraping tasks. Continuity and ease of debugging can be improved by incorporating structured try-except blocks, persistent logging practices, and checkpointing strategies (Ryan, 2020). Additionally, breaking down scraping tasks into smaller, manageable units can prevent memory overflows and increase system fault tolerance.

Adherence to ethical and legal standards in web scraping must also be ensured. Integrating robots.txt compliance into the scraping workflow reduces the risk of violating

site policies. Whenever possible, accessing data through official APIs or through data-sharing agreements is strongly preferred (Singh & Sharma, 2022).

The quality of data can be enhanced by addressing limitations associated with datasets derived from a small number of pages. Future work should explore sources that provide structured or fragmented listings or aggregate data from multiple websites to expand the dataset without overburdening any single server (Kumar & Bansal, 2021).

When processing large volumes of data, effective memory management is essential. Streaming techniques, temporary disc storage, and efficient data formats such as Parquet are recommended. Libraries such as Polars are particularly suitable, as they are optimised for performance and exhibit low memory consumption, enabling multiple ETL tasks to be executed in memory simultaneously (van der Maas, 2023).

9.0 References

Amazon Web Services. (2023). Amazon EC2 features. https://aws.amazon.com/ec2/features/

Faiz, A. (2022, December 12). Selectolax — A better Python web scraping library. Medium. https://medium.com/@afiffaiz/selectolax-a-better-python-web-scraping-library-2a8c1ee5551b

Fischer, B. (2024, December 23). Python advanced: 10 things you can do with Polars (and didn't know about it). Medium. https://captain-solaris.medium.com/python-advanced-10-things-you-can-do-with-polars-and-didnt-know-about-it-cb8c071227ba

Kolli, A. (2024, February 8). Accelerating pandas with Modin: A comprehensive guide. Medium. https://aravindkolli.medium.com/accelerating-pandas-with-modin-a-comprehensive-guide-90697ca4173d

Kumar, R., & Bansal, S. (2021). Efficient web scraping approaches and tools for data extraction: A comprehensive review. International Journal of Web & Semantic Technology, 12(2), 45–59. https://doi.org/10.5121/ijwest.2021.12204

Kumawat, K. (2024, November 23). What is Parsel in Python? Medium. https://medium.com/@kuldeepkumawat195/what-is-parsel-in-python-b76237cb7e6d

Mitchell, R. (2021). Web scraping with Python: Collecting data from the modern web (2nd ed.). O'Reilly Media.

Omisola, I. (2025, February 27). 7 best Python web scraping libraries in 2025. ZenRows. https://www.zenrows.com/blog/python-web-scraping-library

Ryan, D. (2020). Practical Python web scraping: Best practices and techniques. Apress.

Seng, S. A. C. (2025, May 15). HTTPX [Python] | Fast-performing and robust Python library. Apidog: An integrated platform for API design, debugging, development, mock, and testing. https://apidog.com/blog/what-is-httpx/

Shahizan, D. (n.d.). drshahizan [GitHub repository]. GitHub. https://github.com/drshahizan/drshahizan

Singh, A., & Sharma, M. (2022). Ethical issues in web data extraction: Legal challenges and best practices. Journal of Information Ethics, 31(1), 50–65. https://doi.org/10.3172/JIE.31.1.050

van der Maas, R. (2023). Polars user guide: Fast and memory-efficient dataframes in Python. https://pola-rs.github.io/polars-book/

Vishnoi, P. (2023, August 20). What is Dask in Python. Medium. https://premvishnoi.medium.com/what-is-dask-in-python-bc86730ec69e

Whelan, L. (2025, April 8). Web crawling vs. web scraping – What's the difference? SOAX. https://soax.com/blog/web-crawling-vs-web-scraping#:~:text=Web%20crawlers%20visit%20the%20target,the%20information%20to%20be%20scraped

10.0 Appendices

10.1 Appendix A

Table A1 below lists the data attributes extracted from Carlist.my, which form the basis of our data collection and analysis.

Table A1: Overview of Car Data Attributes from Carlist.my

Attribute	Description
Car Name	The full name of the car, including model year, brand, and trim/version details.
Car Brand	The manufacturer or brand of the vehicle.
Car Model	The specific model of the car.
Manufacture Year	The year in which the car was made.
Body Type	The style or design of the car.
Fuel Type	The type of fuel the car uses.
Mileage	The total distance the car has been driven, measured in kilometres (KM).
Transmission	The type of transmission the car uses.
Color	The colour of the car's exterior.
Price	The cost of the car, usually in the local currency.
Installment	The monthly payment if the car is purchased through an instalment plan.
Condition	The state of the car, either new, used or reconditioned.

Seat Capacity	The number of people the car can accommodate, including the driver and passengers.
Location	The geographic location where the car is being sold.
Sales Channel	The type of seller or platform through which the car is being sold.

10.2 Appendix B

Table B1 provides detailed descriptions of the components involved in the system architecture, which includes the web scraping process and big data tools used for data processing, cleaning, transformation, and performance evaluation.

Table B1: Web Scraping and Big Data Processing Architecture Components

Component	Description
Crawler	The web scraping process was initiated by navigating the targeted website, following links, and collecting raw HTML content (Whelan, 2025).
Parser	The raw HTML content was processed to extract key elements such as car name, car model, and car price for further analysis (Whelan, 2025).
Data Collection	The parsed data from individually assigned page ranges was stored in a structured format (CSV file) and later saved in MongoDB for further export and processing.
Data Cleaning, Transformation, and Optimisation	The scraped data was cleaned by handling missing values, duplicates, and inconsistencies. It was then transformed into a suitable format for analysis. During this process, optimisation techniques such as parallel processing and multi-threading were implemented using different libraries (Pandas, Polars, Modin, Dask).
Performance Evaluation	The performance of the entire system was measured and evaluated based on speed, CPU usage, memory usage, and throughput.
End-To-End Flow	A complete pipeline was established, where data flowed from scraping to storage, cleaning, transformation, optimisation, and performance evaluation for efficient data analysis.

Table B2 summarises the libraries and frameworks utilised by each team member during the web scraping and big data processing stages of the project. The tools are categorised based on their respective functionalities to provide an overview of the software stack employed for efficient data acquisition and analysis.

Table B2: Tools and Frameworks Used

Web Crawling & Web Scraping			
Name	**Library (Crawler)**	**Library (Scraper)**	**Description**
Marcus Joey Sayner	httpx	parsel	httpx is a modern, async-capable HTTP client used to fetch web content (Seng, 2025), whereas parsel is used to extract data from HTML/XML using CSS or XPath selectors (Kumawat, 2024).
Muhammad Luqman Hakim bin Mohd Rizaudin	urllib	Selectolax	urllib is a built-in Python module to handle HTTP request (Omisola, 2025), and selectolax is a fast, lightweight HTML parser optimised for speed and large documents.
Camily Tang Jia Lei	urllib	BeautifulSoup	Similarly, urllib handles the retrieval of web pages, and BeautifulSoup parses and extracts structured data from HTML content (Omisola, 2025).
Goh Jing Yang	requests	BeautifulSoup	requests is a simple and widely used HTTP library for making web requests, and BeautifulSoup is used to parse and navigate the HTML content for data

			extraction (Omisola, 2025).
Big Data Processing			
Name	**Library**	**Description**	
Marcus Joey Sayner	Polars	A fast DataFrame library for data manipulation, optimised for performance and scalability using multi-threaded execution (Fischer, 2024).	
Muhammad Luqman Hakim bin Mohd Rizaudin	Modin	A library designed to speed up pandas workflows using built-in parallelisation (Kolli, 2024).	
Camily Tang Jia Lei	Pandas	A popular Python library for data manipulation and analysis, offering efficient data structures like DataFrame and Series to handle and process large datasets (Kolli, 2024).	
Goh Jing Yang	Dask	A flexible parallel computing library that extends pandas-like APIs for analytics that can handle larger-than-memory computations (Vishnoi, 2023).	

10.3 Appendix C

Figures C1 to C4 present the source code implementations for the crawling and scraping methods used by each team member. Each figure corresponds to a specific combination of libraries used for web data extraction from the Carlist.my website.

```python
import httpx
import csv
import time
import json
from parsel import Selector

base_url = "https://www.carlist.my/cars-for-sale/malaysia?page_number={}&page_size=25"
start_page = 1
end_page = 2
car_listings = []
output_file = 'carlist_data.csv'

# Extract from the <script type="application/ld+json"> block
def extract_json_ld(selector):
    scripts = selector.css('script[type="application/ld+json"]::text').getall()
    for script in scripts:
        try:
            data = json.loads(script)
            if isinstance(data, list):
                for d in data:
                    if 'itemListElement' in d:
                        return d['itemListElement']
        except json.JSONDecodeError:
            continue
    return None

# Get mileage from <i class="icon--meter">
def get_mileage(selector):
    meter_icon = selector.css('i.icon--meter')
    if meter_icon:
        return meter_icon.xpath('following-sibling::text()[1]').get(default='').strip()
    return ''

# Scraping function to get the page content
def fetch_page(url):
    response = httpx.get(url)
    return response.text

# Correct full location (e.g., Petaling Jaya, Selangor)
def get_location(selector, car):
    try:
        # Extract the locality and region from the car JSON data
        address_locality = car.get('seller', {}).get('homeLocation', {}).get('address', {}).get('addressLocality', '')
        address_region = car.get('seller', {}).get('homeLocation', {}).get('address', {}).get('addressRegion', '')

        # Combine locality and region to form the location
        if address_locality and address_region:
            location = f"{address_locality}, {address_region}"
        else:
            location = ', '.join([address_locality, address_region]).strip(', ')

        return location
    except Exception as e:
        return ''

# Correct sales agent (e.g., Sales Agent)
def get_sales_channel(selector):
    try:
        # Use CSS selector to find the sales agent text
        sales_info = selector.css('span.c-chip--icon::text').get(default='').strip()
        # Replace "Sales" with "Sales Agent"
        sales_channel = sales_info.replace("Sales", "Sales Agent") if "Sales" in sales_info else sales_info
        return sales_channel
    except Exception as e:
        return ''

# Start timing
start_time = time.time()
```

```python
# Start scraping
for page in range(start_page, end_page + 1):
    url = base_url.format(page)
    print(f"\n🔎 Scraping page {page}: {url}")

    response = httpx.get(url, headers={"User-Agent": "Mozilla/5.0"})
    selector = Selector(response.text)

    articles = selector.css('article.listing')
    json_ld = extract_json_ld(selector)

    if not json_ld:
        print("⚠️ No JSON-LD found.")
        continue

    for i, (article_sel, json_car) in enumerate(zip(articles, json_ld)):
        car = json_car['item']

        name = article_sel.attrib.get('data-title', '')
        brand = article_sel.attrib.get('data-make', '')
        model = article_sel.attrib.get('data-model', '')
        body = article_sel.attrib.get('data-body-type', '')
        transmission = article_sel.attrib.get('data-transmission', '')
        installment = article_sel.attrib.get('data-installment', '')

        year = car.get('vehicleModelDate', '')
        fuel = car.get('fuelType', '')
        color = car.get('color', '')
        price = car.get('offers', {}).get('price', '')
        condition = car.get('itemCondition', '')
        seats = car.get('seatingCapacity', '')

        mileage = get_mileage(article_sel)
        location = get_location(article_sel, car)
        sales_channel = get_sales_channel(article_sel)

        car_listings.append({
            'Car Name': name,
            'Car Brand': brand,
            'Car Model': model,
            'Manufacture Year': year,
            'Body Type': body,
            'Fuel Type': fuel,
            'Mileage': mileage,
            'Transmission': transmission,
            'Color': color,
            'Price': price,
            'Installment': installment,
            'Condition': condition,
            'Seating Capacity': seats,
            'Location': location,
            'Sales Channel': sales_channel
        })

    print(f"✅ Extracted {len(articles)} listings from page {page}")
    time.sleep(2)  # Be kind to the server

# Save to CSV
if car_listings:
    with open(output_file, 'w', newline='', encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=car_listings[0].keys())
        writer.writeheader()
        writer.writerows(car_listings)
    print(f"\n💾 Saved {len(car_listings)} listings to '{output_file}'")
else:
    print("⚠️ No data extracted.")

# End timing
end_time = time.time()
print(f"\n⏱️ Total scraping time: {end_time - start_time:.2f} seconds")
```

Figure C1: Crawling and scraping code using httpx + parsel

```python
import urllib.request
from selectolax.parser import HTMLParser
import csv
import time
import json

# Configuration
base_url = "https://www.carlist.my/cars-for-sale/malaysia?page_number={}&page_size=25"
start_page = 1745    # Starting page number to scrape
end_page = 3488      # Ending page number to scrape
car_listings = []  # List to store car listings data

output_file = 'car_listings.csv'

def get_attr(element, attr_name):
    return element.attributes.get(attr_name, '') if element else ''

def get_text(node):
    return node.text().strip() if node else ''

def get_sales_channel(article):
    dealer_div = article.css_first('div.listing__spec--dealer')
    if dealer_div:
        channel = dealer_div.text().strip()
        # Convert "Sales" to "Sales Agent"
        return channel.replace("Sales", "Sales Agent") if "Sales" in channel else channel
    return ''

def get_mileage(article):
    icon = article.css_first('i.icon--meter')
    if icon and icon.next:
        return icon.next.text().strip()
    return ''

def get_location(article):
    icon = article.css_first('i.icon--location')
    if not icon:
        return ''
    text = ''
    current = icon.next
    while current:
        if current.tag == '-text':
            text += current.text().strip()
        elif current.tag == 'span':
            text += current.text().strip()
        else:
            break
        current = current.next
    return text.strip()

def extract_json_ld(parser):
    for script in parser.css('script[type="application/ld+json"]'):
        try:
            data = json.loads(script.text())
            if isinstance(data, list):
                for d in data:
                    if 'itemListElement' in d:
                        return d['itemListElement']
        except:
            continue
    return None

start_time = time.time()
```

```python
for page_num in range(start_page, end_page + 1):
    url = base_url.format(page_num)
    print(f"\n🔎 Extracting page {page_num} - {url}")

    req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
    html = urllib.request.urlopen(req).read()
    parser = HTMLParser(html)

    articles = parser.css('article.listing')
    ld_json = extract_json_ld(parser)
    if not ld_json:
        print("⚠️ No JSON-LD found on this page.")
        continue

    page_count = 0
    for article, item in zip(articles, ld_json):
        car = item['item']

        name = get_attr(article, 'data-title')
        brand = get_attr(article, 'data-make')
        model = get_attr(article, 'data-model')
        body = get_attr(article, 'data-body-type')
        transmission = get_attr(article, 'data-transmission')
        installment = get_attr(article, 'data-installment')

        mileage = get_mileage(article)
        sales_channel = get_sales_channel(article)  # This now returns "Sales Agent" if "Sales" was found
        location = get_location(article)

        year = car.get('vehicleModelDate', '')
        fuel = car.get('fuelType', '')
        color = car.get('color', '')
        price = car.get('offers', {}).get('price', '')
        condition = car.get('itemCondition', '')
        seats = car.get('seatingCapacity', '')

        car_listings.append({
            'Car Name': name,
            'Car Brand': brand,
            'Car Model': model,
            'Manufacture Year': year,
            'Body Type': body,
            'Fuel Type': fuel,
            'Mileage': mileage,
            'Transmission': transmission,
            'Color': color,
            'Price': price,
            'Installment': installment,
            'Condition': condition,
            'Seating Capacity': seats,
            'Location': location,
            'Sales Channel': sales_channel  # Will show "Sales Agent" instead of "Sales"
        })
        page_count += 1

    time.sleep(2)

if car_listings:
    with open(output_file, 'w', newline='', encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=car_listings[0].keys())
        writer.writeheader()
        writer.writerows(car_listings)
    print(f"\n✅ Saved {len(car_listings)} cars to '{output_file}'")
else:
    print("\n⚠️ No car listings found.")

end_time = time.time()
execution_time = end_time - start_time
print(f"\n🕒 Total execution time: {execution_time:.2f} seconds")
```

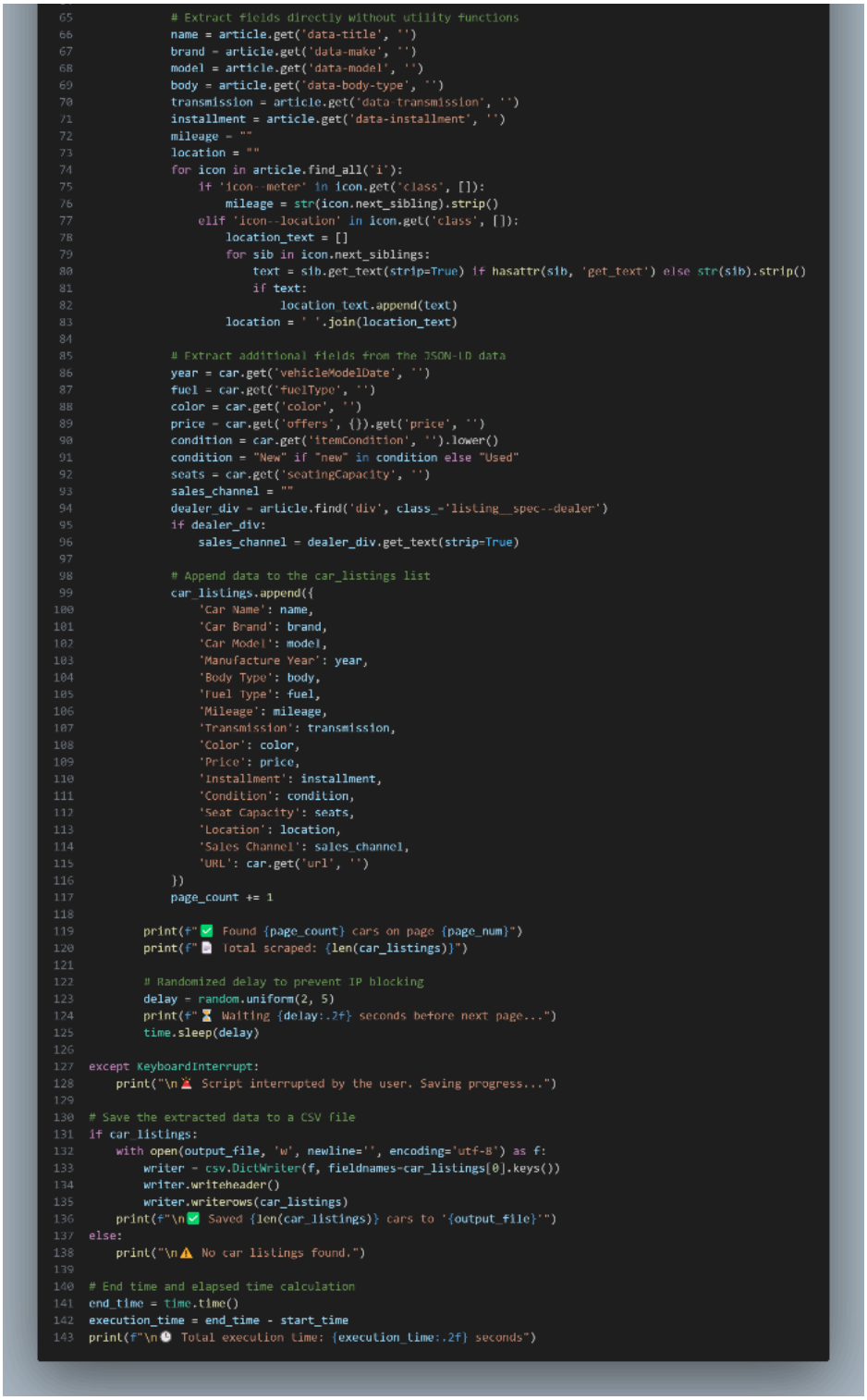
Figure C2: Crawling and scraping code using urllib + Selectolax

```python
import urllib.request
from bs4 import BeautifulSoup, NavigableString
import csv
import time
import json

# Configuration
base_url = "https://www.carlist.my/cars-for-sale/malaysia?page_number={}&page_size=25"
start_page = 5233    # Starting page number to scrape
end_page = 6977      # Ending page number to scrape
car_listings = []  # List to store car listings data

output_file = 'D:/camily_carlist_listings_1.csv'

# Get HTML attribute or empty string if attribute doesn't exist
def get_attr(element, attr_name):
    return element.get(attr_name, '') if element else ''

# Get the first text node inside a <div> (ignores non-text elements)
def get_first_text_node(div):
    for node in div.contents if div else []:
        if isinstance(node, NavigableString) and node.strip():
            return node
    return ''  # Return empty string if no text node is found

# Extract raw mileage information from the article (after the meter icon)
def get_mileage(article):
    icon = article.find('i', class_='icon--meter')  # Find meter icon
    return str(icon.next_sibling) if icon and icon.next_sibling else ''  # Return mileage text

# Extract raw location information from the article (after the location icon)
def get_location(article):
    icon = article.find('i', class_='icon--location')  # Find location icon
    if not icon:
        return ''
    text = ''
    for sib in icon.next_siblings:  # Loop through siblings after icon
        if isinstance(sib, NavigableString):
            text += str(sib)
        elif getattr(sib, 'name', None) == 'span':
            text += sib.get_text()
        else:
            break
    return text  # Return extracted location text

# Pull JSON-LD block containing the car listings from the page
def extract_json_ld(soup):
    for script in soup.find_all('script', type='application/ld+json'):
        try:
            data = json.loads(script.string)  # Parse the JSON-LD data
            if isinstance(data, list):
                for d in data:
                    if 'itemListElement' in d:
                        return d['itemListElement']  # Return list of car listings
        except:
            continue
    return None  # Return None if no valid JSON-LD is found

# Start time
start_time = time.time()

# Main scraping loop
for page_num in range(start_page, end_page + 1):
    url = base_url.format(page_num)  # Generate URL for the current page
    print(f"\n🔎 Extracting page {page_num} - {url}")
```

```
    # Load the page and parse it with BeautifulSoup
    req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
    html = urllib.request.urlopen(req).read()
    soup = BeautifulSoup(html, 'html.parser')

    # Find all articles (car listings) and extract JSON-LD data
    articles = soup.find_all('article', class_='listing')
    ld_json = extract_json_ld(soup)
    if not ld_json:
        print("⚠️ No JSON-LD found on this page.")
        continue

    page_count = 0
    # Loop through articles and corresponding JSON-LD data
    for article, item in zip(articles, ld_json):
        car = item['item']  # Extract JSON details for the car

        # Extract fields from the <article> element's attributes (HTML-based fields)
        name = get_attr(article, 'data-title')
        brand = get_attr(article, 'data-make')
        model = get_attr(article, 'data-model')
        body = get_attr(article, 'data-body type')
        transmission = get_attr(article, 'data-transmission')
        installment = get_attr(article, 'data-installment')

        # Extract exactly what is displayed on the page (visible text-based fields)
        mileage = get_mileage(article)
        sales_channel = get_first_text_node(article.find('div', class_='listing__spec--dealer'))
        location = get_location(article)

        # Extract additional fields from the JSON-LD data
        year = car.get('vehicleModelDate', '')
        fuel = car.get('fuelType', '')
        color = car.get('color', '')
        price = car.get('offers', {}).get('price', '')
        condition = car.get('itemCondition', '')
        seats = car.get('seatingCapacity', '')

        # Append the extracted data to the car_listings list
        car_listings.append({
            'Car Name': name,
            'Car Brand': brand,
            'Car Model': model,
            'Manufacture Year': year,
            'Body Type': body,
            'Fuel Type': fuel,
            'Mileage': mileage,
            'Transmission': transmission,
            'Color': color,
            'Price': price,
            'Installment': installment,
            'Condition': condition,
            'Seating Capacity': seats,
            'Location': location,
            'Sales Channel': sales_channel
        })
        page_count += 1

    print(f"✅ Found {page_count} cars on page {page_num}")
    print(f"📄 Total scraped: {len(car_listings)}")
    time.sleep(2)  # Sleep to avoid making too many requests in a short time

# Save the extracted data to a CSV file
if car_listings:
    with open(output_file, 'w', newline='', encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=car_listings[0].keys())
        writer.writeheader()  # Write the column headers
        writer.writerows(car_listings)  # Write the car listings data
    print(f"\n✅ Saved {len(car_listings)} cars to '{output_file}'")
else:
    print("\n⚠️ No car listings found.")

# End time and elapsed time calculation
end_time = time.time()
execution_time = end_time - start_time
print(f"\n🕒 Total execution time: {execution_time:.2f} seconds")
```

Figure C3: Crawling and scraping code using urllib + BeautifulSoup

```python
import requests
from bs4 import BeautifulSoup
import csv
import json
import time
import random

# Configuration
base_url = "https://www.carlist.my/cars-for-sale/malaysia?page_number={}&page_size=25"
start_page = 3471
end_page = 5205
car_listings = []
output_file = 'car_listings.csv'
max_retries = 3

# Start time
start_time = time.time()

# Main scraping loop
try:
    for page_num in range(start_page, end_page + 1):
        url = base_url.format(page_num)
        print(f"\n🔎 Extracting page {page_num} - {url}")

        # Retry mechanism for network errors
        retries = 0
        while retries < max_retries:
            try:
                headers = {"User-Agent": "Mozilla/5.0"}
                response = requests.get(url, headers=headers, timeout=10)
                response.raise_for_status()
                soup = BeautifulSoup(response.content, 'html.parser')
                break
            except requests.exceptions.RequestException as e:
                retries += 1
                print(f"⚠️ Network error: {e}. Retrying ({retries}/{max_retries})...")
                time.sleep(2)
        else:
            print(f"❌ Failed to fetch page {page_num} after {max_retries} retries. Skipping...")
            continue

        # Find all articles (car listings) and extract JSON-LD data
        articles = soup.find_all('article', class_='listing')
        ld_json = None
        for script in soup.find_all('script', type='application/ld+json'):
            try:
                data = json.loads(script.string)
                if isinstance(data, list):
                    for d in data:
                        if 'itemListElement' in d:
                            ld_json = d['itemListElement']
                            break
            except Exception as e:
                print(f"⚠️ Error parsing JSON-LD: {e}")
                continue

        if not ld_json:
            print(f"⚠️ No JSON-LD found on page {page_num}. Skipping...")
            continue

        page_count = 0
        for article, item in zip(articles, ld_json):
            car = item['item']
```



```
            # Extract fields directly without utility functions
            name = article.get('data-title', '')
            brand = article.get('data-make', '')
            model = article.get('data-model', '')
            body = article.get('data-body-type', '')
            transmission = article.get('data-transmission', '')
            installment = article.get('data-installment', '')
            mileage = ""
            location = ""
            for icon in article.find_all('i'):
                if 'icon--meter' in icon.get('class', []):
                    mileage = str(icon.next_sibling).strip()
                elif 'icon--location' in icon.get('class', []):
                    location_text = []
                    for sib in icon.next_siblings:
                        text = sib.get_text(strip=True) if hasattr(sib, 'get_text') else str(sib).strip()
                        if text:
                            location_text.append(text)
                    location = ' '.join(location_text)

            # Extract additional fields from the JSON-LD data
            year = car.get('vehicleModelDate', '')
            fuel = car.get('fuelType', '')
            color = car.get('color', '')
            price = car.get('offers', {}).get('price', '')
            condition = car.get('itemCondition', '').lower()
            condition = "New" if "new" in condition else "Used"
            seats = car.get('seatingCapacity', '')
            sales_channel = ""
            dealer_div = article.find('div', class_='listing__spec--dealer')
            if dealer_div:
                sales_channel = dealer_div.get_text(strip=True)

            # Append data to the car_listings list
            car_listings.append({
                'Car Name': name,
                'Car Brand': brand,
                'Car Model': model,
                'Manufacture Year': year,
                'Body Type': body,
                'Fuel Type': fuel,
                'Mileage': mileage,
                'Transmission': transmission,
                'Color': color,
                'Price': price,
                'Installment': installment,
                'Condition': condition,
                'Seat Capacity': seats,
                'Location': location,
                'Sales Channel': sales_channel,
                'URL': car.get('url', '')
            })
            page_count += 1

        print(f"✅ Found {page_count} cars on page {page_num}")
        print(f"📄 Total scraped: {len(car_listings)}")

        # Randomized delay to prevent IP blocking
        delay = random.uniform(2, 5)
        print(f"⏳ Waiting {delay:.2f} seconds before next page...")
        time.sleep(delay)

except KeyboardInterrupt:
    print("\n🚨 Script interrupted by the user. Saving progress...")

# Save the extracted data to a CSV file
if car_listings:
    with open(output_file, 'w', newline='', encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=car_listings[0].keys())
        writer.writeheader()
        writer.writerows(car_listings)
    print(f"\n✅ Saved {len(car_listings)} cars to '{output_file}'")
else:
    print("\n⚠️ No car listings found.")

# End time and elapsed time calculation
end_time = time.time()
execution_time = end_time - start_time
print(f"\n🕒 Total execution time: {execution_time:.2f} seconds")
```

Figure C4: Crawling and scraping code using requests + BeautifulSoup

10.4 Appendix D

Table D**1** provides a comprehensive overview of the data cleaning procedures applied to the car listings dataset. Each entry in the table outlines the specific objective and detailed steps followed to ensure the dataset's accuracy, consistency, and analytical readiness.

Table D1: Summary of Data Cleaning Methods

Combine Columns: 'Seat Capacity' and 'Seating Capacity'	
Objective	To merge redundant or synonymous columns for uniformity of the data.
Steps	1. The dataset was observed to contain two columns, namely 'Seat Capacity' and 'Seating Capacity', both representing the same attribute.
	2. To eliminate redundancy and ensure consistency, these columns were consolidated into a single column labelled 'Seat Capacity', which was designated as the primary reference for seating information.
	3. The following procedure was applied: ● For each row, if the value in 'Seat Capacity' was null but 'Seating Capacity' contained a valid entry, the null value was filled with the corresponding value from 'Seating Capacity'. ● If a value already existed in 'Seat Capacity', it was retained and given precedence.
	4. After completing the merge, the 'Seating Capacity' column was removed from the dataset.
Remove Unwanted Column(s): 'URL'	
Objective	To eliminate unnecessary or noisy data that does not contribute to the

	analysis.
Steps	1. The column ‘URL’ was identified as lacking analytical relevance.
	2. It was observed that the column contained lengthy strings referring to external web resources, which were: ● Not utilised in analysis or visualizations, ● Not consistently present across all entries, ● Potentially misleading or distracting.
	3. Consequently, the ‘URL’ column was removed from the dataset to reduce dimensionality and streamline the analytical process.
String Cleaning and Formatting (Clean ‘Installment’ Column’)	
Objective	To remove currency symbols and units from the values to convert them into numeric form.
Steps	1. The “RM” currency symbol was removed.
	2. Commas (,) used as thousand separators were eliminated.
	3. The “/month” suffix was stripped from the values.
	4. The cleaned strings were converted into numeric data.
	5. As a result, the ‘Installment’ column now contains standardised numeric values suitable for aggregation and statistical analysis
String Cleaning and Formatting (Clean ‘Condition’ Column)	
Objective	To extract and simplify metadata-style URLs into clear categorical labels.
Steps	1. Strings were parsed to extract the main condition label from URLs such as "https://schema.org/UsedCondition", retaining only values like "Used" or "New".

	2. Any trailing "Condition" suffixes were removed.
	3. The column was finalised with clean categorical values suitable for filtering and classification.
String Cleaning and Formatting (Clean 'Sales Channel' Column)	
Objective	To standardise and extract meaningful labels from long-form sales channel strings.
Steps	1. Regular expressions were applied to extract relevant categories such as "Sales Agent" or "Dealer".
	2. Entries not matching the expected pattern were either removed or replaced with null values.
	3. The column now contains standardised sales channel identifiers for reliability.
String Cleaning and Formatting (Clean 'Mileage' Column)	
Objective	To normalise mileage values into consistent categorical bins for analysis.
Steps	1. The "K KM" unit suffix was removed from all entries.
	2. Range values (e.g., "5 - 10") were parsed and normalised.
	3. Single numeric values (e.g., "12") were binned into 5K-step ranges (e.g., "10 - 15").
	4. The column now presents uniform mileage categories for improved aggregation and visualisation.
Replace / Drop Null Values	
Objective	To ensure data completeness by removing rows with missing values in key columns.

Steps	1. The following primary columns were selected: 'Body Type', 'Fuel Type', 'Mileage', 'Sales Channel', and 'Seat Capacity'.
	2. These columns were checked for missing (NaN) values
	3. Rows containing nulls in any of the selected columns were removed using .dropna().
	4. Only complete rows were retained, eliminating potential bias from imputation or scattered missing data.
Rename Columns (Add Units for Clarity)	
Objective	To improve readability by appending measurement units to column names.
Steps	1. Key numeric columns were identified and renamed as follows: ● 'Mileage' → 'Mileage (K KM)' ● 'Price' → 'Price (RM)' ● 'Installment' → 'Installment (RM)'
	2. The .rename() method with inplace=True was used to apply changes directly to the original DataFrame.
Drop Duplicated Rows	
Objective	To eliminate redundant records and avoid distortion in the analysis.
Steps	1. Duplicate rows were identified using .duplicated() across all columns.
	2. Exact duplicates were removed using .drop_duplicates().
	3. Care was taken to retain semantically valid duplicates, ensuring meaningful data was preserved.

10.5 Appendix E

Table E1 summarises the key performance optimisation strategies used by each data processing library.

Table E1: Overview of Data Processing Libraries

Library	**Explanation**
Pandas	● Operates in a single-threaded manner by default. ● No built-in parallelism is provided. ● Act as the control group to evaluate the performance improvements of other libraries.
Polars	● Operates using **multi-threading** to parallelise data operations at the thread level. ● Built in Rust, providing faster performance and lower memory overhead compared to Pandas.
Modin	● Enables automatic **parallelism** using the Dask execution engines. ● Spreads DataFrame operations across all CPU cores to speed up processing on multi-core systems. ● Requires minimal code changes from Pandas.
Dask	● Applies task-based **parallelism** and lazy computation graphs to break down large operations into smaller, manageable tasks. ● Optimises memory usage by chunking data and only loading necessary partitions into memory.

Table E2 provides a summary of optimisation tasks and the corresponding code snippets implemented across the data processing libraries.

Table E2: Code Overview of Each Library

Library	**Code**
Combine Dataset	
Figure E1 shows the code used to combine multiple scraped datasets from group members into a single DataFrame using pandas. Four CSV files, each scraped separately, are read and merged into one dataset. The combined result is then saved as a new CSV file. Subsequently, a connection is established to a local MongoDB database, and the combined data is inserted into a collection named “listings”.	

```python
import pandas as pd
from pymongo import MongoClient

# List of group member scraaping files
files = [
    "camily_carlist_listings.csv",
    "luqman_carlist_listings.csv",
    "jingyang_carlist_listings.csv",
    "marcus_carlist_listings.csv"
]

# Read and combine datasets
dfs = [pd.read_csv(file) for file in files]
combined_df = pd.concat(dfs, ignore_index=True)

# Save combined dataset
combined_df.to_csv("carlist_combined.csv", index=False)

# Output shape and preview
print("Combined dataset shape:", combined_df.shape)
print(combined_df.head(5))

# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017")
db = client["carlist_db"]
collection = db["listings"]

# Insert combined data into MongoDB
collection.insert_many(combined_df.to_dict(orient="records"))
print("✅ Data inserted into MongoDB successfully.")
```

Figure E1: Dataset Combination Code

Install and Import Libraries	

<table>
<tr>
<td>Pandas</td>
<td>Figure E2 shows the essential libraries for the Pandas implementation where:

pandas: For reading, merging, and manipulating structured tabular data.
numpy: To support numerical operations.
re: For performing regular expression-based string cleaning and extraction.
time: To record execution time for performance evaluation.
psutil: A system-level information such as memory and CPU usage.
os: For interacting with the operating system environment.
tracemalloc: To trace memory allocation during code execution.
Pymongo: Communicate with a MongoDB database to store the processed data.

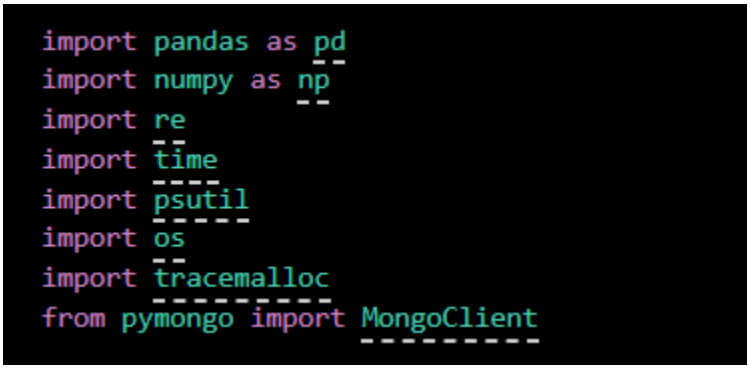
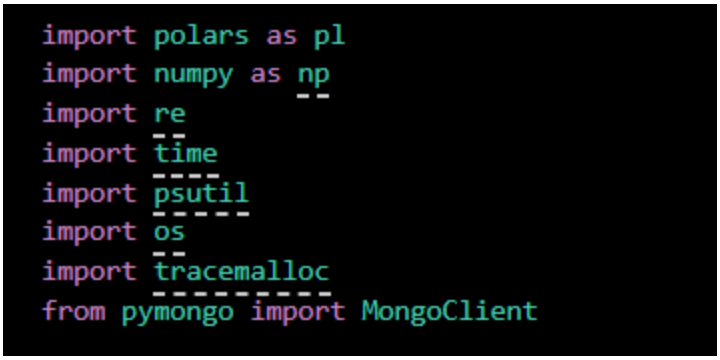
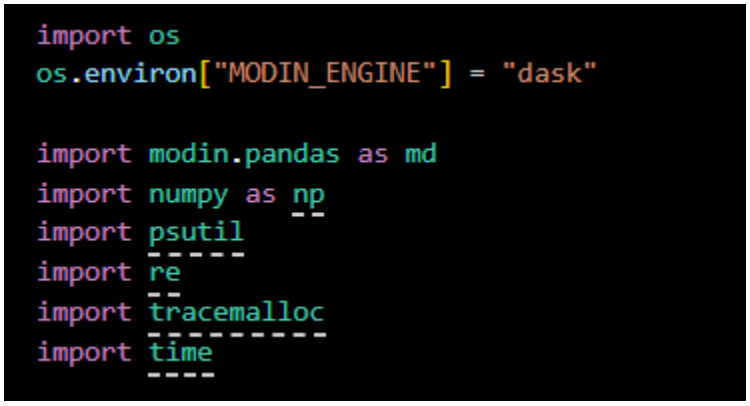


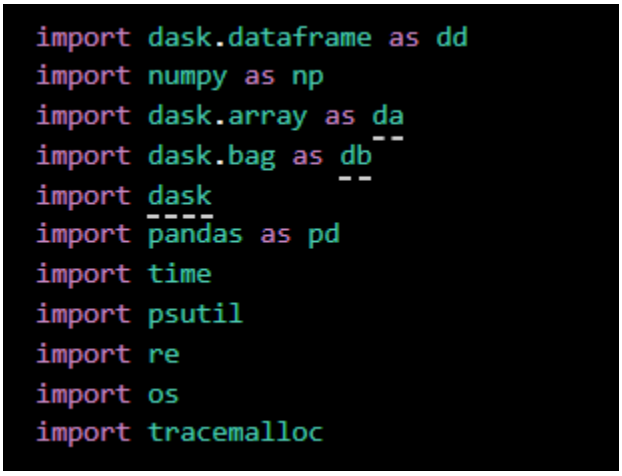
Figure E2: Pandas Code for Install and Import Libraries</td>
</tr>
<tr>
<td>Polars</td>
<td>Figure E3 shows the essential libraries for the Polars implementation where:

Polars: For fast and memory-efficient DataFrame operations, optimised for performance on large datasets.
numpy, re, time, psutil, os, and tracemalloc: Same purposes as in the Pandas version, enabling numerical operations, system monitoring, and memory tracking.
pymongo: For inserting the final dataset into MongoDB.

</td>
</tr>
</table>

	 Figure E3: Polars Code for Install and Import Libraries
Modin	Figure E4 shows the essential libraries for the Modin implementation where: ● The environment variable MODIN_ENGINE is set to "dask" to specify Dask as the backend engine for distributed execution. ● **modin.pandas:** Acts as a drop-in replacement for Pandas but enables parallel processing. ● **numpy, psutil, re, tracemalloc, and time:** For data manipulation, system monitoring, and performance tracking. ● **Os:** To set the environment configuration.  Figure E4: Modin Code for Install and Import Libraries
Dask	Figure E5 shows the essential libraries for the Dask implementation where: ● **dask.dataframe:** To perform scalable DataFrame operations on large datasets.

- **dask.array:** Allow distributed processing of numerical and unstructured data respectively.
- **dask:** Provides additional task scheduling and parallelism utilities.
- **pandas:** Used for fallback operations or conversion between formats.
- **numpy, time, psutil, re, os, and tracemalloc:** Imported for standard data handling, timing, system monitoring, and memory tracking.



```
import dask.dataframe as dd
import numpy as np
import dask.array as da
import dask.bag as db
import dask
import pandas as pd
import time
import psutil
import re
import os
import tracemalloc
```

Figure E5: Dask Code for Install and Import Libraries

General Code

Figure E6 and E7 illustrates the general performance tracking code used consistently throughout the project for all libraries during the import, cleaning and export phases. At the beginning of execution, the process ID is retrieved using psutil.Process(os.getpid()), and memory tracking is initiated with tracemalloc.start(). The start time, CPU time, and CPU usage percentage are recorded to establish a baseline. After the operation (e.g., import or cleaning), the end time and CPU time are captured, and both current and peak memory usage are measured using tracemalloc.get_traced_memory() before stopping the tracker.

Performance metrics are then calculated:

- **Elapsed time:** The total real-world (wall clock) time taken to complete the operation, measured from start to finish.
- **CPU time:** The actual amount of CPU processing time consumed during execution, calculated as the difference in user CPU time before and after the task and converted into milliseconds.
- **CPU usage:** Average percentage of CPU resources utilised during the process, measured over a one-second sampling interval to reflect system load.
- **Throughput:** The processing efficiency by calculating the number of rows handled per second, obtained by dividing the total rows processed by the elapsed time.
- **Current memory usage:** The amount of memory occupied by the Python process at the time of measurement.
- **Peak memory usage:** The maximum memory consumption during execution, highlighting the highest resource demand encountered.

```
# Setup
process_import_pandas = psutil.Process(os.getpid())
tracemalloc.start()

# Record start time and CPU usage
start_time_import_pandas = time.perf_counter()
start_cpu_time_import_pandas = psutil.cpu_times().user  # Start CPU time
start_cpu_percent_import_pandas = psutil.cpu_percent(interval=None)
```

Figure E6: General Code Used Across all Libraries (Initial Setup)

```python
# Record end time, memory, and CPU usage
end_time_import_pandas = time.perf_counter()
end_cpu_time_import_pandas = psutil.cpu_times().user  # End CPU time
current_mem_import_pandas, peak_mem_import_pandas = tracemalloc.get_traced_memory()
tracemalloc.stop()

# CPU percent used during execution
cpu_usage_import_pandas = psutil.cpu_percent(interval=1)  # sampled over 1 second

# Calculate metrics
elapsed_time_import_pandas = end_time_import_pandas - start_time_import_pandas
cpu_time_import_pandas = (end_cpu_time_import_pandas - start_cpu_time_import_pandas) * 1000  # CPU time in ms
total_rows_import_pandas = len(df_pandas)
throughput_import_pandas = total_rows_import_pandas / elapsed_time_import_pandas if elapsed_time_import_pandas > 0 else 0

# Output preview
print("✅ Data loaded from MongoDB")
print(df_pandas.head(3))

# Performance summary
print("\n============== Performance Summary ==============")
print(f"Total rows loaded     : {total_rows_import_pandas}")
print(f"Wall Time (Elapsed)   : {elapsed_time_import_pandas:.4f} seconds")
print(f"CPU Time              : {cpu_time_import_pandas:.0f} ms")
print(f"CPU Usage             : {cpu_usage_import_pandas:.2f}%")
print(f"Throughput            : {throughput_import_pandas:.2f} rows/sec")
print(f"Current Memory (Python): {current_mem_import_pandas / 1e6:.2f} MB")
print(f"Peak Memory (Python)   : {peak_mem_import_pandas / 1e6:.2f} MB")
print("=================================================")
```

Figure E7: General Code Used Across all Libraries (Calculation)

Import Data

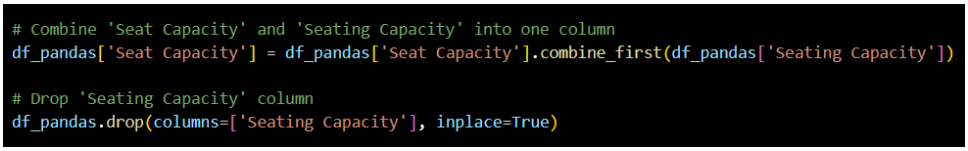
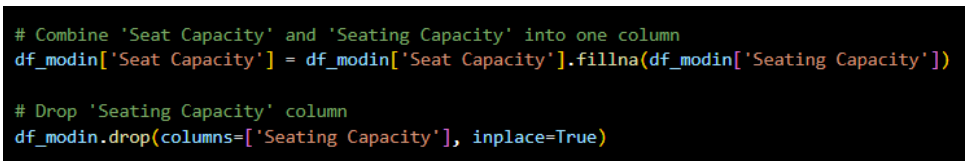
Pandas	Figure E8 shows how data is imported from MongoDB into a Pandas DataFrame. The data is first retrieved as a list of dictionaries from the MongoDB collection. This list is then converted directly into a Pandas DataFrame. Afterward, the MongoDB-specific _id column is removed since it is not needed for further processing.

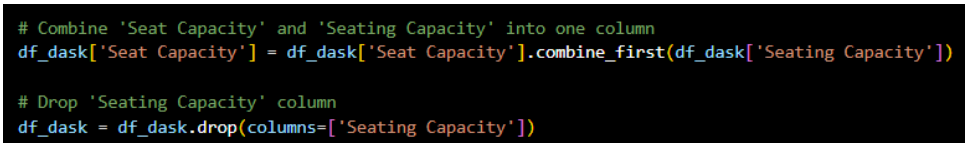
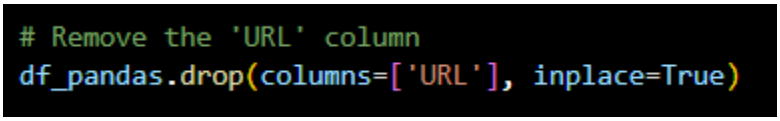
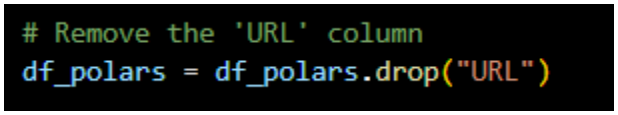
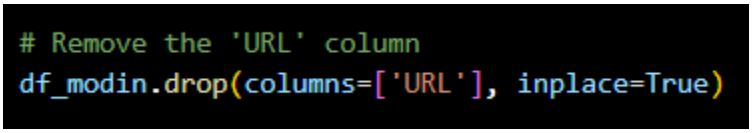
```python
# === Read from MongoDB into pandas DataFrame ===
data_import_pandas = list(collection.find())
df_pandas = pd.DataFrame(data_import_pandas)
df_pandas.drop(columns=['_id'], inplace=True)
# ================================================
```

Figure E8: Pandas Code for Import Data

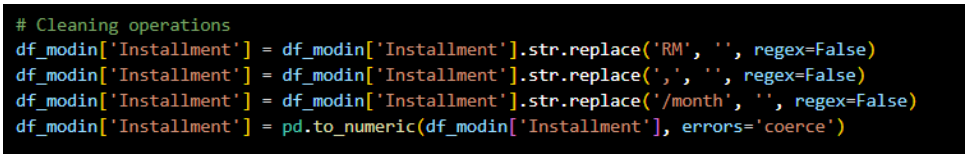
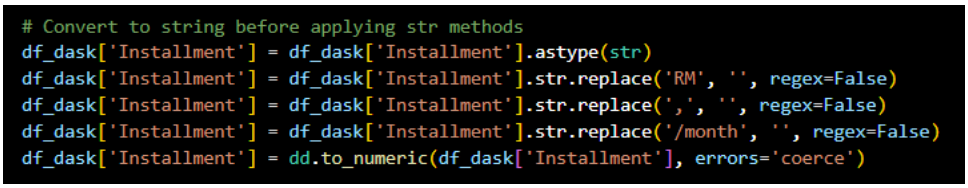
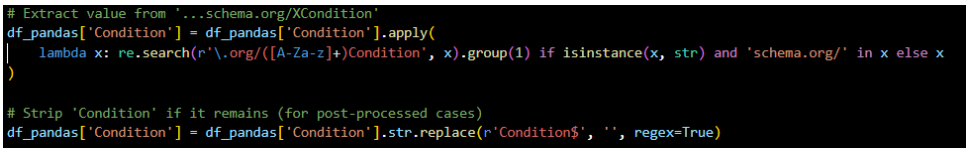
Polars	Figure E9 illustrates the import of data from MongoDB into a Polars DataFrame. Similar to Pandas, the data is fetched as a list of dictionaries. The Polars DataFrame is then created by inferring the schema from a large

	sample of rows to ensure proper data typing. The _id column generated by MongoDB is dropped to clean the dataset. `# === Read from MongoDB into Polars DataFrame ===` `data_import_polars = list(collection.find())` `df_polars = pl.DataFrame(data_import_polars, infer_schema_length=100000)` `df_polars = df_polars.drop("_id")` `# ==========` Figure E9: Polars Code for Import Data
Modin	The procedure for loading data into a Modin DataFrame is shown in Figure E10. In order to speed up Pandas operations through parallel processing, the MongoDB documents are first gathered as a list and then transformed into a Modin DataFrame. After that, the _id column is eliminated to get the data ready for analysis. `# === Read from MongoDB into Modin DataFrame ===` `data_import_modin = list(collection.find())` `df_modin = md.DataFrame(data_import_modin)` `df_modin.drop(columns=['_id'], inplace=True)` `# ==========` Figure E10: Modin Code for Import Data
Dask	The process of reading data into a Dask DataFrame is shown in Figure E11. Initially, MongoDB data is retrieved and transformed into a Pandas DataFrame. Dask is then used to divide this Pandas DataFrame into several smaller pieces so that parallel computation is possible. To maintain the data's relevance for upcoming tasks, the _id column is removed. `# === Read from MongoDB into Dask DataFrame ===` `data = list(collection.find())  # Fetch all documents as a list of dicts` `df_dask = dd.from_pandas(pd.DataFrame(data), npartitions=4)` `df_dask = df_dask.drop(columns=['_id'])  # Drop the '_id' column` `# ==========` Figure E11: Dask Code for Import Data
Combine Columns	

Pandas	Figure E12 shows the Pandas approach to combining the Seat Capacity and Seating Capacity columns. It uses the combine_first() method, which efficiently fills missing values in the first column using values from the second column.  Figure E12: Pandas Code for Combine Columns
Polars	Figure E13 demonstrates the process in Polars, where the coalesce() function is used after converting "NaN" strings into proper nulls. This method is optimised with built-in multithreading, allowing it to perform faster on large datasets without any extra configuration.  Figure E13: Polars Code for Combine Columns
Modin	Figure E14 displays the Modin approach, using the fillna() function to fill missing values in Seat Capacity with values from Seating Capacity. This is similar to Pandas but benefits from Modin's parallel execution for faster performance on large data.  Figure E14: Modin Code for Combine Columns

<table>
<tr><td>Dask</td><td>Figure E15 shows the operation using Dask, also using combine_first() like Pandas. In this context, it runs in parallel on local cores, giving moderate speed improvements.
 
Figure E15: Dask Code for Combine Columns</td></tr>
<tr><td colspan="2">Remove Unwanted Column(s)</td></tr>
<tr><td>Pandas</td><td>Figure E16 shows the removal of the URL column using Pandas. The drop() function is used with inplace=True, and the operation runs on a single thread.
 
Figure E16: Pandas Code for Remove Unwanted Column(s)</td></tr>
<tr><td>Polars</td><td>Figure E17 demonstrates the same column removal in Polars. The drop() method is applied, and the operation benefits from Polars' built-in multithreading, making it faster on larger datasets.
 
Figure E17: Polars Code for Remove Unwanted Column(s)</td></tr>
<tr><td>Modin</td><td>Figure E18 illustrates how the URL column is removed in Modin using the same syntax as Pandas. However, unlike Pandas, Modin executes the operation in parallel across CPU cores, providing performance gains.
 
Figure E18: Modin Code for Remove Unwanted Column(s)</td></tr>
</table>

Dask	Figure E19 shows the URL column being dropped in Dask. The operation runs in parallel on partitions. `# Remove the 'URL' column` `df_dask = df_dask.drop(columns=['URL'])` Figure E19: Modin Code for Remove Unwanted Column(s)
String Cleaning and Formatting (Clean Installment Column)	
Pandas	Figure E20 shows the cleaning process of the Installment column using Pandas. The string replacement operations and numeric conversion are performed sequentially. `# Cleaning operations` `df_pandas['Installment'] = df_pandas['Installment'].str.replace('RM', '', regex=False)` `df_pandas['Installment'] = df_pandas['Installment'].str.replace(',', '', regex=False)` `df_pandas['Installment'] = df_pandas['Installment'].str.replace('/month', '', regex=False)` `df_pandas['Installment'] = pd.to_numeric(df_pandas['Installment'], errors='coerce')` Figure E20: Pandas Code for Clean Installment Column
Polars	Figure E21 shows how Polars cleans the Installment column with optimised execution. The string replacements and type casting are combined within a single expression using the with_columns() method. `# Clean the 'Installment' column: remove 'RM', ',', '/month', strip spaces, and convert to int` `df_polars = df_polars.with_columns(` `  pl.col("Installment")` `  .str.replace_all("RM", "")` `  .str.replace_all(",", "")` `  .str.replace_all("/month", "")` `  .str.strip_chars()` `  .cast(pl.Int64)` `)` Figure E21: Polars Code for Clean Installment Column
Modin	Figure E22 shows the Modin implementation for cleaning the Installment column. Here, the string manipulations and numeric conversion benefit from Modin's parallel execution framework, which splits the data and processes it across multiple CPU cores.

	 Figure E22: Modin Code for Clean Installment Column
Dask	Figure E23 shows the cleaning of the Installment column using Dask. The DataFrame is partitioned, and string replacements along with the numeric conversion are applied in parallel over these partitions.  Figure E23: Dask Code for Clean Installment Column
String Cleaning and Formatting (Extract Clean Values from Condition Column)	
Pandas	Figure E24 shows the Pandas method where the regex extraction and string replacement are applied using .apply() and .str.replace() respectively. Both operations run sequentially without parallelisation, so no specific optimisation is applied in this code.  Figure E24: Pandas Code for Clean Condition Column
Polars	Figure E25 demonstrates Polars' approach where string extraction and replacement are done via fully vectorised expressions inside .with_columns().

	``python # Vectorized: Extract text before 'Condition' after the last '/' (e.g., 'NewCondition' → 'New') df_polars = df_polars.with_columns(pl.col("Condition") .str.extract(r"/([A-Za-z]+)Condition", 1) .fill_null(pl.col("Condition").str.replace(r"Condition$", "", literal=False)) .alias("Condition")) ``` Figure E25: Polars Code for Clean Condition Column
Modin	Figure E26 uses Modin to perform regex extraction with .apply() and string replacement with .str.replace(). The optimisation occurs because Modin distributes these Pandas-like operations across multiple CPU cores, enabling parallel execution transparently. ```python # First: Extract value from '...schema.org/XCondition' df_modin['Condition'] = df_modin['Condition'].apply(lambda x: re.search(r'\.org/([A-Za-z]+)Condition', x).group(1) if isinstance(x, str) and 'schema.org/' in x else x) # Second: Strip 'Condition' if it remains (for post-processed cases) df_modin['Condition'] = df_modin['Condition'].str.replace(r'Condition$', '', regex=True) ``` Figure E26: Modin Code for Clean Condition Column
Dask	Figure E27 applies regex extraction via .apply() with the meta parameter to define output type, allowing Dask to parallelise this operation on each DataFrame partition. The subsequent string replacement runs in parallel across these partitions, providing moderate speedup over sequential processing. ```python # First: Extract value from '...schema.org/XCondition' df_dask['Condition'] = df_dask['Condition'].apply(lambda x: re.search(r'\.org/([A-Za-z]+)Condition', x).group(1) if isinstance(x, str) and 'schema.org/' in x else x, meta=('Condition', 'object')) # Second: Strip 'Condition' if it remains (for post-processed cases) df_dask['Condition'] = df_dask['Condition'].str.replace(r'Condition$', '', regex=True) ``` Figure E27: Dask Code for Clean Condition Column
String Cleaning and Formatting (Clean Sales Channel)	
Pandas	Figure E28 uses Pandas' .str.extract() method to apply regex extraction on the "Sales Channel" column. ```python # Extract Sales Channel (either 'Sales Agent' or 'Dealer') df_pandas['Sales Channel'] = df_pandas['Sales Channel'].str.extract(r'^(Sales Agent|Dealer)') ```

	Figure E28: Pandas Code for Clean Sales Channel Column
Polars	Figure E29 shows Polars performing regex extraction using the vectorised .str.extract() expression within .with_columns(). Figure E29: Polars Code for Clean Sales Channel Column
Modin	Figure E30 applies .str.extract() in Modin, which internally parallelises the operation across multiple CPU cores by distributing the Pandas-like computation, resulting in faster execution compared to vanilla Pandas. Figure E30: Modin Code for Clean Sales Channel Column
Dask	Figure E31 demonstrates Dask's use of .str.extract() on its partitioned DataFrame, where the extraction runs in parallel on each partition, leveraging Dask's partitioned task scheduling for improved speed on large data. Figure E31: DaskCode for Clean Sales Channel Column
String Cleaning and Formatting (Clean Mileage)	
Pandas	Figure E32 shows Pandas cleaning mileage by first removing the substring "K KM" with a vectorised .str.replace() and then applying a custom Python function via .apply(). The .apply() runs sequentially on the entire column, so the performance is limited by single-thread execution.

```python
# Extract either "Sales Agent" or "Dealer" using regex
df_polars = df_polars.with_columns(
    pl.col("Sales Channel")
    .str.extract(r"^(Sales Agent|Dealer)", 1)
    .alias("Sales Channel")
)
```

```python
# Extract Sales Channel (either 'Sales Agent' or 'Dealer')
df_modin['Sales Channel'] = df_modin['Sales Channel'].str.extract(r'^(Sales Agent|Dealer)')
```

```python
# Extract Sales Channel (either 'Sales Agent' or 'Dealer')
df_dask['Sales Channel'] = df_dask['Sales Channel'].str.extract(r'^(Sales Agent|Dealer)', expand=False)
```

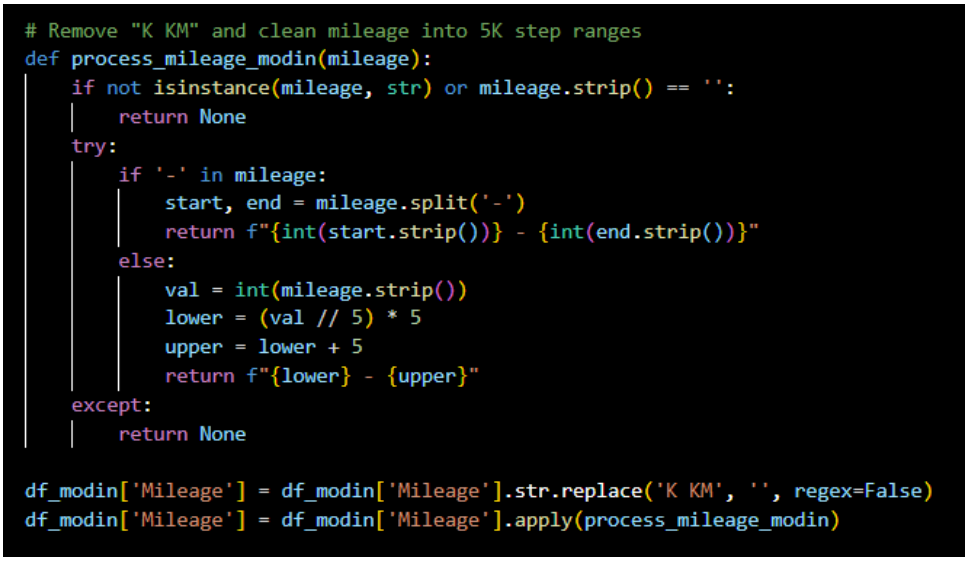
```python
# Remove "K KM" and clean mileage into 5K step ranges
def process_mileage_pandas(mileage):
    if not isinstance(mileage, str) or mileage.strip() == '':
        return None
    try:
        if '-' in mileage:
            start, end = mileage.split('-')
            return f"{int(start.strip())} - {int(end.strip())}"
        else:
            val = int(mileage.strip())
            lower = (val // 5) * 5
            upper = lower + 5
            return f"{lower} - {upper}"
    except:
        return None

df_pandas['Mileage'] = df_pandas['Mileage'].str.replace('K KM', '', regex=False)
df_pandas['Mileage'] = df_pandas['Mileage'].apply(process_mileage_pandas)
```

Figure E32: Pandas Code for Clean Mileage Column

Polars

Figure E33 uses Polars' .map_elements() to apply a custom cleaning function on each element of the "Mileage" column. Polars optimises this by executing the function in a parallel and compiled manner internally, giving it better speed than pure Python .apply() in Pandas.

```python
# Define a function to clean mileage and convert into 5K step range
def clean_mileage(val):
    if val is None or str(val).strip() == '':
        return None
    val = str(val).replace('K KM', '').strip()
    try:
        if '-' in val:
            start, end = val.split('-')
            return f"{int(start.strip())} - {int(end.strip())}"
        else:
            num = int(val)
            lower = (num // 5) * 5
            upper = lower + 5
            return f"{lower} - {upper}"
    except:
        return None

# Apply the cleaning using map_elements
df_polars = df_polars.with_columns(
    pl.col("Mileage")
    .map_elements(clean_mileage, return_dtype=pl.Utf8)
    .alias("Mileage")
)
```

	Figure E33: Polars Code for Clean Mileage Column
Modin	Figure E34 applies the same approach as Pandas but leverages Modin's parallelised .apply(). Modin partitions the DataFrame and distributes the .apply() function across CPU cores, enabling concurrent execution and improved performance over standard Pandas.  Figure E34: Modin Code for Clean Mileage Column
Dask	Figure E35 performs a two-step process: a vectorised .str.replace() to remove "K KM", followed by a .apply() with a custom function executed on each Dask partition. Dask's task scheduler runs these .apply() functions in parallel across partitions, accelerating the processing on large datasets.

```python
# Remove "K KM" and clean mileage into 5K step ranges
def process_mileage_modin(mileage):
    if not isinstance(mileage, str) or mileage.strip() == '':
        return None
    try:
        if '-' in mileage:
            start, end = mileage.split('-')
            return f"{int(start.strip())} - {int(end.strip())}"
        else:
            val = int(mileage.strip())
            lower = (val // 5) * 5
            upper = lower + 5
            return f"{lower} - {upper}"
    except:
        return None

df_modin['Mileage'] = df_modin['Mileage'].str.replace('K KM', '', regex=False)
df_modin['Mileage'] = df_modin['Mileage'].apply(process_mileage_modin)
```

```python
# Remove "K KM"
df_dask['Mileage'] = df_dask['Mileage'].str.replace('K KM', '', regex=False)

# Clean and standardize mileage into 5K step ranges
def process_mileage_dask(mileage):
    if not isinstance(mileage, str) or mileage.strip() == '':
        return None
    try:
        if '-' in mileage:
            start, end = mileage.split('-')
            return f"{int(start.strip())} - {int(end.strip())}"
        else:
            val = int(mileage.strip())
            lower = (val // 5) * 5
            upper = lower + 5
            return f"{lower} - {upper}"
    except:
        return None

df_dask['Mileage'] = df_dask['Mileage'].apply(process_mileage_dask, meta=('x', 'object'))
```

Figure E35: Dask Code for Clean Mileage Column

Replace Null Values

Pandas

Figure E36 shows Pandas replacing empty strings and "-" with np.nan using vectorisation. .replace() calls are efficient built-in string operations. The subsequent .dropna() call removes rows with missing values in specified columns, operating in memory and sequentially.

```python
# Convert to np.nan
df_pandas.replace(r'^\s*$', np.nan, regex=True, inplace=True)  # Empty and whitespace-only strings
df_pandas.replace('-', np.nan, inplace=True)  # "-"

# Check null
df_pandas.isnull().sum()
```

```python
# Drop rows with missing values in specific columns
df_pandas.dropna(subset=['Body Type', 'Fuel Type', 'Mileage', 'Sales Channel', 'Seat Capacity'], inplace=True)
```

Figure E36: Pandas Code for Replace Null Values

Polars

Figure E37 uses Polars' lazy evaluation and expression API to replace empty, "-", or "NaN" strings with null values in all string columns via vectorisation .when().then().otherwise() expressions. The .drop_nulls() method then drops rows with nulls efficiently using Polars' optimised execution engine, which is highly parallelised and memory efficient.

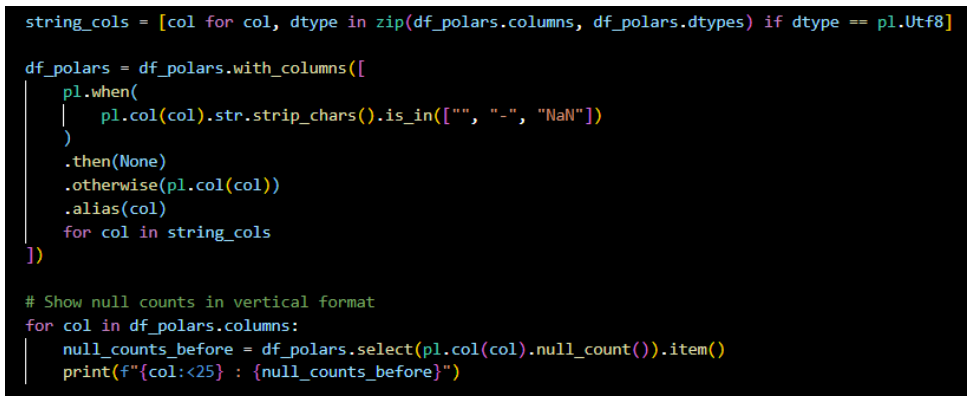
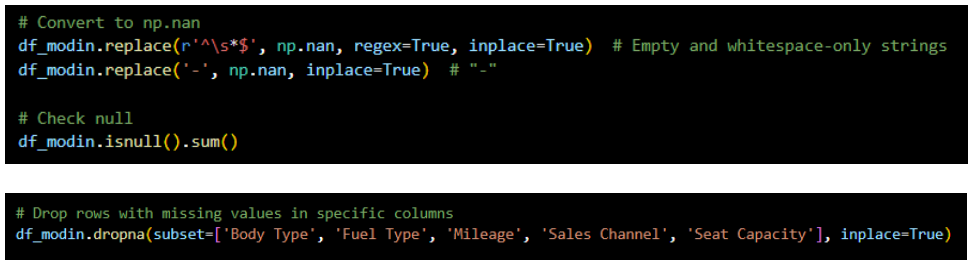
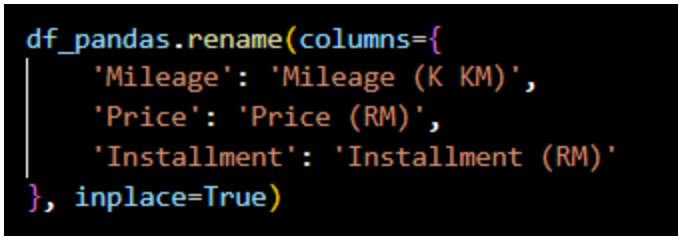
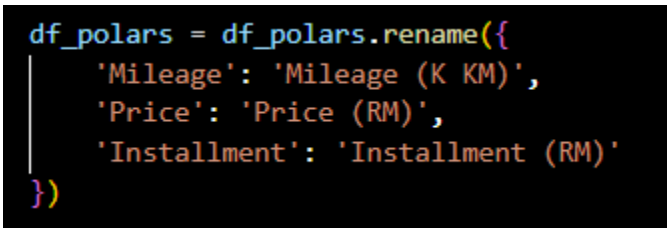
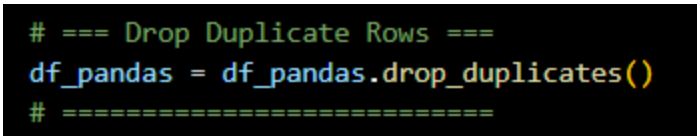
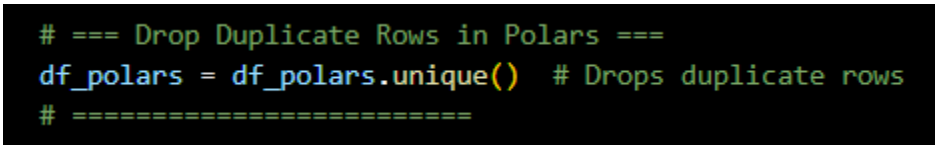
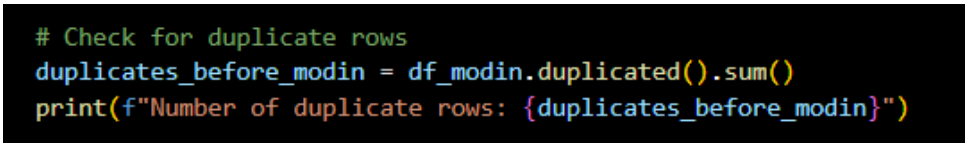
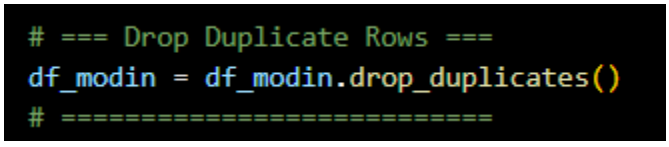
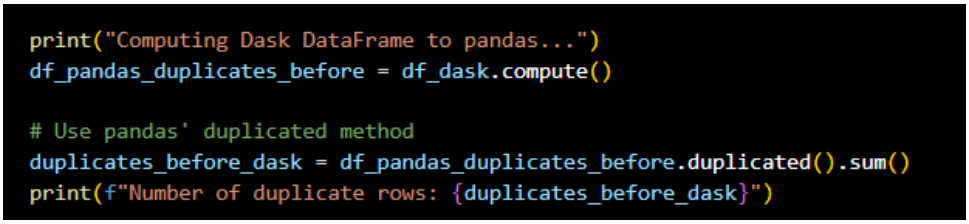
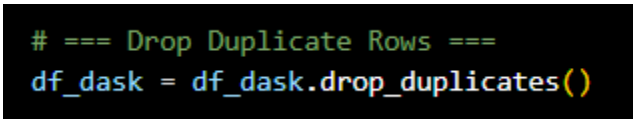
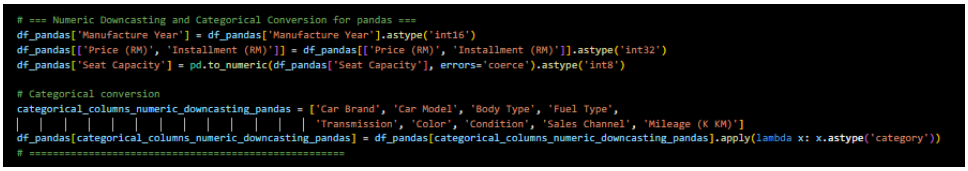
	 Figure E37: Polars Code for Replace Null Values
Modin	Figure E38 applies Pandas-like .replace() with regex support and .dropna() on Modin DataFrame. Modin optimises these operations by splitting data into partitions and executing replacements and null drops concurrently across multiple cores, speeding up these typically expensive row filtering steps.  Figure E38: Modin Code for Replace Null Values
Dask	Figure E39 performs .replace() operations to convert blanks and "-" to np.nan, then .dropna() to remove incomplete rows on Dask DataFrame. Dask parallelises these steps by partitioning data and executing the replacements and null drops across partitions concurrently, then triggers computation with .compute() for aggregated results.

	 Figure E39: Dask Code for Replace Null Values
Rename Columns	
Pandas	Figure E40 illustrates the column renaming in Pandas. The .rename() method with inplace=True is applied, which modifies the DataFrame directly in memory. This method is efficient for smaller to medium datasets, as it avoids creating a copy of the DataFrame and quickly updates column labels.  Figure E40: Pandas Code for Rename Columns
Polars	Figure 5.2.41 shows the renaming process in Polars. Since Polars operates on immutable DataFrames, .rename() returns a new DataFrame with the updated column names. This fits Polars' optimised execution model, allowing the operation to be part of a fast, chainable query pipeline suitable for large datasets.  Figure E41: Polars Code for Rename Columns

Modin	Figure E42 demonstrates the renaming method used by Modin. It applies .rename() with inplace=True, similar to Pandas, but internally Modin distributes the task across multiple cores or nodes, enabling parallelised column renaming. This parallelism improves performance when handling large-scale data. `df_modin.rename(columns={ 'Mileage': 'Mileage (K KM)', 'Price': 'Price (RM)', 'Installment': 'Installment (RM)' }, inplace=True)` Figure E42: Modin Code for Rename Columns
Dask	Figure E43 presents the renaming step in Dask. The .rename() method returns a lazy DataFrame with new column names, deferring actual computation. This lazy evaluation allows Dask to optimise the execution plan and efficiently handle renaming as part of a larger distributed workflow before triggering .compute(). `df_dask = df_dask.rename(columns={ 'Mileage': 'Mileage (K KM)', 'Price': 'Price (RM)', 'Installment': 'Installment (RM)' })` Figure E43: Dask Code for Rename Columns
Drop Duplicated Rows	
Pandas	Figure E44 illustrates how Pandas handles duplicate detection and removal using in-memory operations optimised for single-threaded performance. Pandas applies vectorised methods but can be limited by memory when working with very large datasets. `# Check for duplicate rows` `duplicates_before_pandas = df_pandas.duplicated().sum()` `print(f"Number of duplicate rows: {duplicates_before_pandas}")`

	 Figure E44: Pandas Code for Drop Duplicated Rows
Polars	Figure E45 shows Polars using its built-in unique() method, which is implemented in Rust and optimised for parallel execution. This allows Polars to efficiently drop duplicates with lower memory usage and faster speed compared to pandas.  Figure E45: Polars Code for Drop Duplicated Rows
Modin	Figure E46 depicts Modin's approach, which distributes the duplicate check and removal tasks across multiple cores or nodes using Ray or Dask backends. This parallelisation provides scalability and improved runtime on large datasets relative to pandas.   Figure E46: Modin Code for Drop Duplicated Rows
Dask	Figure E47 demonstrates Dask's method, which computes the lazy DataFrame into pandas to leverage pandas' duplicate detection capabilities.

	  Figure E47: Dask Code for Drop Duplicated Rows
Optimise Memory Usage by Using Efficient Data Types	
Pandas	Figure E48 illustrates the numeric downcasting and categorical conversion process applied to the pandas DataFrame. The numeric columns 'Manufacture Year', 'Price (RM)', 'Installment (RM)', and 'Seat Capacity' were downcast to smaller integer types to reduce memory usage. Additionally, categorical columns such as 'Car Brand', 'Body Type', and 'Fuel Type' were converted to the category dtype to optimise storage and improve performance in subsequent operations.  Figure E48: Pandas Code for Optimise Memory Usage by Using Efficient Data Types
Polars	Figure E49 depicts the implementation of numeric downcasting and categorical conversion in the Polars DataFrame. The relevant numeric columns were cast to smaller integer types using Polars' casting methods. For categorical data, each specified column was cast to the categorical type, facilitating more efficient memory consumption and faster query performance.

```python
# === Numeric Downcasting and Categorical Conversion for Polars ===
df_polars = df_polars.with_columns([
    pl.col("Manufacture Year").cast(pl.Int16),
    pl.col("Price (RM)").cast(pl.Int32),
    pl.col("Installment (RM)").cast(pl.Int32),
    pl.col("Seat Capacity").cast(pl.Int8),
])

# Categorical conversion
categorical_columns_numeric_downcasting_polars = ['Car Brand', 'Car Model', 'Body Type', 'Fuel Type',
                                                  'Transmission', 'Color', 'Condition', 'Sales Channel', 'Mileage (K KM)']
for col in categorical_columns_numeric_downcasting_polars:
    df_polars = df_polars.with_columns([
        pl.col(col).cast(pl.Categorical)
    ])
```

Figure E49: Polars Code for Optimise Memory Usage by Using Efficient Data Types

Modin

Figure E50 shows the numeric downcasting and categorical conversion carried out on the Modin DataFrame. Similar to pandas, key numeric columns were downcasted to reduced integer types, and selected categorical columns were converted to the category datatype.

```python
# === Numeric Downcasting and Categorical Conversion for Modin ===
df_modin['Manufacture Year'] = df_modin['Manufacture Year'].astype('int16')
df_modin[['Price (RM)', 'Installment (RM)']] = df_modin[['Price (RM)', 'Installment (RM)']].astype('int32')
df_modin['Seat Capacity'] = pd.to_numeric(df_modin['Seat Capacity'], errors='coerce').astype('int8')

# Categorical conversion
categorical_columns_numeric_downcasting_modin = ['Car Brand', 'Car Model', 'Body Type', 'Fuel Type',
                                                 'Transmission', 'Color', 'Condition', 'Sales Channel', 'Mileage (K KM)']
df_modin[categorical_columns_numeric_downcasting_modin] = df_modin[categorical_columns_numeric_downcasting_modin].apply(lambda x: x.astype('category'))
# ================================================
```

Figure E50: Modin Code for Optimise Memory Usage by Using Efficient Data Types

Dask

Figure E51 presents the numeric downcasting and categorical conversion performed on the Dask DataFrame. The numeric columns were downcasted to smaller integer types, while categorical columns were individually converted to the category dtype.

```python
# Numeric downcasting for Dask
df_dask['Manufacture Year'] = df_dask['Manufacture Year'].astype('int16')
df_dask[['Price (RM)', 'Installment (RM)']] = df_dask[['Price (RM)', 'Installment (RM)']].astype('int32')
df_dask['Seat Capacity'] = dd.to_numeric(df_dask['Seat Capacity'], errors='coerce').astype('int8')

# Categorical conversion
categorical_columns_numeric_downcasting_dask = ['Car Brand', 'Car Model', 'Body Type', 'Fuel Type',
                                                'Transmission', 'Color', 'Condition', 'Sales Channel', 'Mileage (K KM)']
# Convert each column to category individually
for col in categorical_columns_numeric_downcasting_dask:
    df_dask[col] = df_dask[col].astype('category')
```

Figure E51: Dask Code for Optimise Memory Usage by Using Efficient Data Types

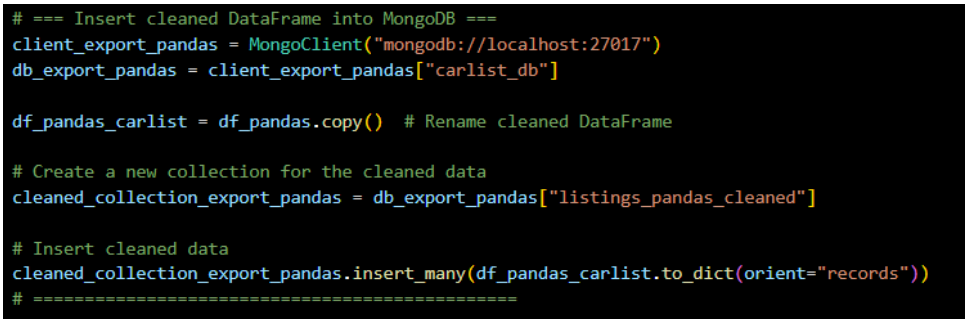
<table>
<tr><th colspan="2">Export Cleaned Data to MongoDB</th></tr>
<tr><td>Pandas</td><td>Figure E52 shows the cleaned pandas DataFrame being converted to dictionaries and inserted into MongoDB. The process used direct conversion for efficient data export.
 

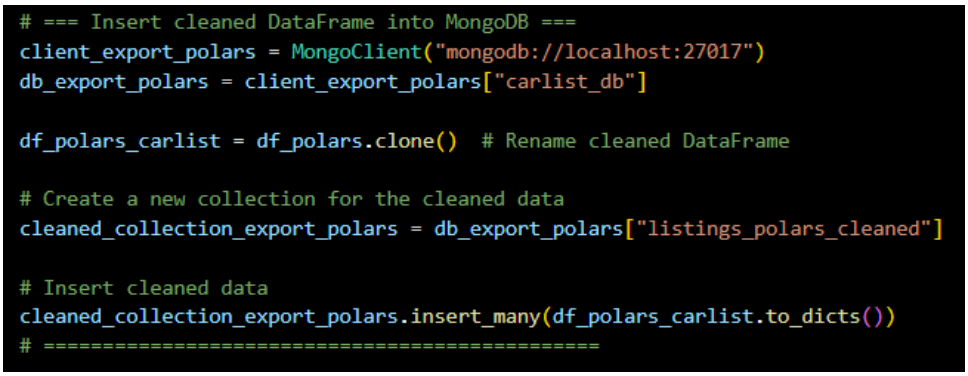
Figure E52: Pandas Code for Export Cleaned Data to MongoDB</td></tr>
<tr><td>Polars</td><td>Figure E53 illustrates the cleaned Polars DataFrame cloned and converted to dictionaries before insertion into MongoDB, leveraging Polars' fast data handling.
 

Figure E53: Polars Code for Export Cleaned Data to MongoDB</td></tr>
<tr><td>Modin</td><td>Figure E54 depicts the cleaned Modin DataFrame copied and transformed to dictionary format for batch insertion into MongoDB, benefiting from Modin's parallel processing.</td></tr>
</table>

<table>
<tr><td></td><td>

```
# === Insert cleaned DataFrame into MongoDB ===
client_export_modin = MongoClient("mongodb://localhost:27017")
db_export_modin = client_export_modin["carlist_db"]

df_modin_carlist = df_modin.copy()  # Rename cleaned DataFrame

# Create a new collection for the cleaned data
cleaned_collection_export_modin = db_export_modin["listings_modin_cleaned"]

# Insert cleaned data
cleaned_collection_export_modin.insert_many(df_modin_carlist.to_dict(orient="records"))
# ============================================
```

Figure E54: Modin Code for Export Cleaned Data to MongoDB
</td></tr>
<tr><td>Dask</td><td>Figure E55 presents the cleaned Dask DataFrame being computed to pandas, then converted and inserted into MongoDB, accounting for Dask's lazy evaluation.

```
# === Insert cleaned Dask DataFrame into MongoDB ===

client_export_dask = MongoClient("mongodb://localhost:27017")
db_export_dask = client_export_dask["carlist_db"]

# Compute Dask DataFrame to Pandas DataFrame for MongoDB insertion
df_dask_carlist = df_dask.compute()  # Convert Dask DataFrame to Pandas DataFrame

# Create a new collection for the cleaned data
cleaned_collection_export_dask = db_export_dask["listings_dask_cleaned"]

# Insert cleaned data into MongoDB
cleaned_collection_export_dask.insert_many(df_dask_carlist.to_dict(orient="records"))
# ============================================
```

Figure E55: Dask Code for Export Cleaned Data to MongoDB
</td></tr>
</table>

10.6 Appendix F

Table F1 summarises the performance metrics collected during data processing tasks for each library, including time, CPU usage, throughput, and memory consumption.

Table 6.2.1: Performance Metrics Comparison Across Data Processing Libraries

	Metrics	**Elapsed Time (Wall Time) (s)**	**CPU Time (ms)**	**CPU Usage (%)**	**Throughput (rows/sec)**	**Current Memory Usage (MB)**	**Peak Memory Usage (MB)**
Steps	**Library**						
Import Data	**Pandas**	10.7775	14828	13.00	16158.60	448.21	524.84
	Polars	18.5478	23859	13.00	9389.27	424.51	436.65
	Modin	62.6945	187359	24.60	2777.75	442.99	542.54
	Dask	45.9962	90578	14.80	3786.18	450.60	807.31
Combine Columns	**Pandas**	0.0824	62	8.70	2112196.89	22.34	23.71
	Polars	0.0393	94	15.40	4429832.37	0.04	0.06
	Modin	0.4683	641	28.00	371837.94	0.83	1.24
	Dask	0.1055	78	7.20	1650515.34	0.20	0.65
Remove Unwanted Column (s)	**Pandas**	0.0405	31	14.80	4298248.86	20.92	20.94
	Polars	0.0011	0	6.80	154333569.63	0.00	0.02
	Modin	0.0048	0	7.50	36599974.78	0.01	0.03
	Dask	0.1541	78	8.00	1130332.57	0.13	0.65
Clean Installme	**Pandas**	1.8746	2047	9.80	92898.42	19.13	34.51

nt Column	**Polars**	0.0556	94	6.90	3129715.50	0.02	0.04
	Modin	3.9595	5094	5.50	43983.06	3.11	19.68
	Dask	0.1376	109	8.10	1265861.43	0.26	0.66
Extract Clean Values from Condition Column	**Pandas**	1.2767	1375	8.50	13604.88	9.13	16.64
	Polars	0.0710	62	11.00	2453134.74	0.02	0.29
	Modin	0.7658	1391	9.50	227422.13	1.31	1.75
	Dask	0.3295	375	11.80	528492.96	0.28	0.78
Extract Clean Values from Sales Channel Column	**Pandas**	1.2360	1578	7.30	140902.18	9.84	27.68
	Polars	0.0217	16	9.50	8010594.42	0.02	0.04
	Modin	1.7670	2922	12.50	98556.84	1.67	19.02
	Dask	0.3289	703	13.60	529517.81	0.20	0.72
Clean Mileage Column Into 5K Step Ranges	**Pandas**	2.5527	2719	11.90	68222.64	9.75	26.75
	Polars	2.5342	2922	12.50	68719.17	0.28	0.54
	Modin	0.8994	2109	20.60	193626.21	1.28	1.89
	Dask	0.3569	203	10.50	487891.87	0.26	0.75
Replace Null Values	**Pandas**	0.2154	281	14.80	808379.11	21.13	29.60
	Polars	0.0112	0	7.40	15585287.27	0.01	0.03
	Modin	4.0161	6125	16.80	43362.71	6.92	8.30

	Dask	21.1509	40359	19.90	8233.69	5.05	344.34
Rename Columns	**Pandas**	0.0050	156	12.80	33195271.89	0.00	0.02
	Polars	0.0046	0	12.50	36049846.76	0.01	0.03
	Modin	0.0045	0	18.70	36660828.65	0.02	0.04
	Dask	20.1589	33641	12.00	8160.95	4.83	344.34
Drop Duplicated Rows	**Pandas**	0.4250	406	11.10	378294.51	20.65	30.84
	Polars	0.0435	141	15.80	3696134.72	0.01	0.03
	Modin	6.2243	9531	44.70	25833.47	8.46	8.97
	Dask	0.0668	344	11.50	2407074.39	0.09	0.33
Optimise Memory Usage by Using Efficient Data Types	**Pandas**	1.3797	2297	11.10	116544046	3.53	21.75
	Polars	0.0452	94	8.50	3553856.58	0.02	0.04
	Modin	13.9932	20641	18.90	11490.86	14.53	15.20
	Dask	0.0840	78	19.40	1913473.14	0.34	0.36
Export Cleaned Data to MongoDB	**Pandas**	18.4362	31219	19.90	8721.67	14.62	184.51
	Polars	11.4875	14891	14.40	13997.33	0.50	269.53
	Modin	19.8898	26250	13.20	8084.24	4.73	202.58
	Dask	41.0053	60609	9.80	3921.30	24.40	344.57

10.7 Appendix G

The full project code and dataset are available at the following GitHub repository:
https://github.com/Jingyong14/HPDP02/tree/main/2425/project/p1/Group%201