UNIVERSITI TEKNOLOGI MALAYSIA

FACULTY OF COMPUTING

20242025/2

SECP3133 – HIGH PERFORMANCE DATA PROCESSING

SECTION 01

PROJECT 1 : Optimizing Multithreaded Web Scraping (NST News) Analysis and Data Processing with Pandas, Dask, and Polars

LECTURER NAME :

PROF. MADYA. TS. DR. MOHD SHAHIZAN BIN OTHMAN

NAME	MATRIC ID
LEE YIK HONG	A21BE0376
WONG JUN JI	A22EC0117
CHE MARHUMI BIN CHE AB RAHIM	A22EC0147

Prepared by : Group A Submission date : 16/5/2025

1.0 Introduction

1.1 Project Background

In today's digital age, the sheer amount of information available online makes it both an opportunity and a challenge to collect and make sense of that data efficiently. This project sets out to build a fast and reliable web scraping system that can gather large amounts of news articles from the New Straits Times (NST)—specifically from the Business, World, and ASEAN sections. Because the NST website loads content dynamically using JavaScript, basic scraping tools aren't sufficient. To overcome this, we used Selenium—a tool that can interact with dynamic content like a real user would—along with multithreading to speed up the entire process.

The system was designed to be both efficient and scalable. It uses multiple browser instances running at the same time (via worker threads), with each one responsible for scraping a different page. All the extracted article data—such as the category, headline, summary, and date—is safely collected and saved into a structured CSV file for further processing.

After collecting the data, we focused on cleaning and formatting it using three different Python libraries: Pandas, Dask, and Polars. These tools each have their own strengths, especially when it comes to handling large datasets. We evaluated their performance based on several factors: how long they took to process the data, how much CPU and memory they used, and how many records they could handle per second (throughput).

This report will walk through how the system was built, how the data was collected and processed, and how each of the libraries performed. We'll also discuss some of the challenges we faced along the way and what could be improved in the future.

1.2 Objectives

- The primary objectives of this project are as follows:
- To design and implement a multithreaded web scraping system capable of collecting over 100,000 structured news records.
- To apply high-performance data processing techniques using libraries such as Pandas, Dask, and Polars.
- To analyze and compare the efficiency of different processing pipelines based on execution time, resource utilization, and throughput.

- To store the final dataset in a structured format suitable for further analysis and research purposes..

1.3 Target website and data to be extracted

The scraping process targets three sections of the New Straits Times website:

1. https://www.nst.com.my/world
2. https://www.nst.com.my/business
3. https://www.nst.com.my/world/region (ASEAN)

Each section contains dynamically loaded news articles that require automated interaction for extraction. The system focuses on extracting the following structured data fields:

Category – The news section (e.g., Business, World, ASEAN)

Headline – The title of the article

Summary – A brief summary of the content

Date – The date and time of publication

The extracted dataset serves as the basis for evaluating data cleaning, transformation, and optimization strategies across multiple Python libraries tailored for high-performance data processing.

2.0 System Design & Architecture

2.1 Description of architecture

The architecture of our high-performance web scraping system for NST new follows a distinct multi-stage process, emphasizing a centralized storage for raw data before cleaning and optimization. The system's components and data flow are illustrated in the diagram below:

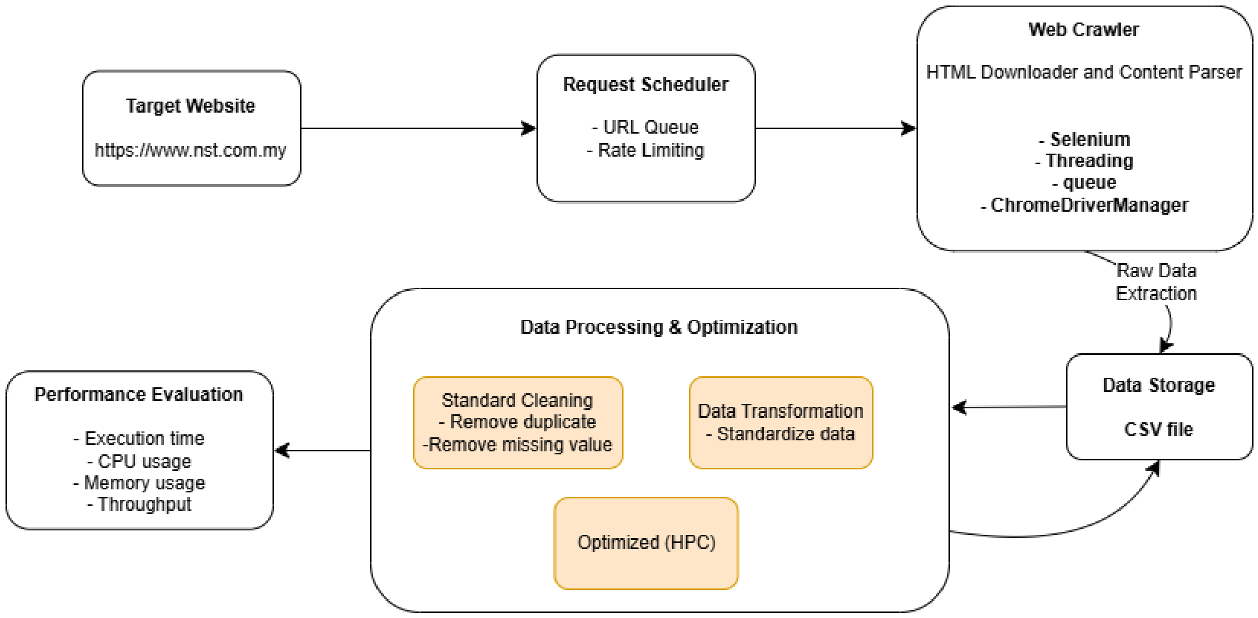


Figure 2.1.1 Architecture Diagram

Figure 2.1.1 is the architecture diagram that illustrates the workflow of our web scraping and data processing used in this project. The architecture of this web scraping script is designed to efficiently collect article metadata from the New Straits Times (NST) by utilizing multithreading and Selenium automation. At its core, the script adopts a producer-consumer model, where a queue holds the list of page numbers to be scrapped, and multiple worker threads consume tasks from this queue concurrently. Each worker initializes its own headless Chrome browser instance using Selenium, configured to run without a graphical interface and with images, stylesheets, and fonts disabled to improve performance. The (scrape_with_driver) function is responsible for navigating to each page, waiting for article elements to load, and extracting information such as the article’s headline, category, summary, and publication date. All successfully scraped data is collected into a shared results list, which is thread-safe in CPython due to the Global Interpreter Lock (GIL). Once all threads complete their tasks, the data is written to a CSV file for storage and analysis. This architecture significantly reduces total scraping time by allowing multiple pages to be processed in parallel while maintaining modularity, scalability, and resilience through robust error handling and efficient browser management. For the performance evaluation part, we implement some performance metric to measure our scrawler ability and performance with the time it takes for the scrawler to scrap the data, CPU usage, Memory usage, and Throughput. We visualize these performance metrics into graph using matplotlib in python for more details visualization

2.2 Tools and Frameworks used

Our project utilizes a range of tools and frameworks to facilitate efficient web scraping.

Tools/Library	Description
Selenium	Needed when content loads dynamically or requires interaction (e.g., click to expand menu or paginate).
Chrome WebDriver	A standalone server used to automate the Google Chrome browser with the Selenium WebDriver
Threading	Allow multiple tasks (like scraping different pages) to run at the same time, making the process faster and more efficient compared to running tasks one after another.
CSV	Act as a database to store the data
Python	Programming language used to write web scraping scripts
Pandas	Python library for data manipulation and analysis. It was used for cleaning, transforming, and processing the extracted data effectively
Polars	A high-performance DataFrame library written in Rust, serving as an alternative to Pandas when working with larger datasets that require more speed.
Dask	A flexible library for parallel computing in Python
psutil	Used for retrieving information about running processes and system utilization.
Matplotlib	Library use to plot performance metric

2.3 Role of Team Members

Name	Role	Task Description
LEE YIK HONG	Group Leader, Evaluator	- Choose a Malaysian website - Coordinated group tasks - Develop scraping of items by subcategory
WONG JUN JI	Architect, Coder	- Design system architecture - Handle performance evaluation and result compilation
CHE MARHUMI	Coder	- Develop and test optimized web crawler - Ensure accurate data extraction - Develop optimized data cleaning

Table 2.3.1 Role of Team Members

3.0 Data Collection

3.1 Crawling Method

- Pagination
  - Implements sequential pagination by generating URLs page number by using a loop in the main() function.
  - Pages are fetched using the URL pattern until the defined page limit is reached.
- Concurrency
  - Utilizes the threading module for multithreaded scraping, improving performance by parallelizing the workload.
  - A Queue is used to manage and distribute page numbers across multiple threads.
- Browser Optimization
  - Applies a headless Chrome browser to reduce rendering overhead and improve speed.
- Dynamic Content Handling
  - Uses WebDriverWait along with expected_conditions to wait for specific elements (.article-teaser) to load before scraping, ensuring the scraper handles dynamically rendered content properly.

3.2 Number of Records Collected

Crawler	Records Collected
Crawler 1 (Business section)	30,000
Crawler 2 (Asean section)	4,399
Crawler 3 (World section)	71,981
Total =	106,380

Each page contains 20 articles that will be extracted.

Table 3.2.1 show the section of news collected by 3 crawler

3.2 Ethical Considerations

To ensure responsible data collection, the web scraper was developed following ethical guidelines:

- The target website, NST news, is a public news portal, and only data that was openly accessible was collected. The web scraping process strictly adhered to the website's robots.txt to avoid accessing restricting areas, adhering to the site usage policy.
- No private data or personal information was collected or stored. The content collected consisted only of article metadata like titles, publish dates, categories, and URLs which were already publicly accessible from the publisher.
- The scrapers were set with proper rate-limiting and delays to avoid putting too much load on the host server. The approach reduced opportunities for service interruption or damage to the operationality of the website.
- The data collected was solely utilized for research and academic purposes for the undertaking of the project. There is no redistribution of the scraped content as intended.

4.0 Data Processing

4.1 Cleaning Methods

a. Handle Duplicate Data:
Some articles may appear multiple times, especially when tagged under different categories. To confirm the uniqueness of data, duplicated are removed from data set

library	Code snippet
Pandas & Dask	df = df.drop_duplicates()
Polars	df = df.unique()

b. Handle Missing and Empty Data:
Missing attributes in key columns can reduce data quality. These are filtered or filled to ensure consistency

Library	Code snippet

Pandas & Dask	`df['headline'] =df['headline'].astype(str).str.strip()`
	`df = df[df['headline'] != ""]`
	`df['summary'] = df['summary'].fillna("").astype(str).str.strip()`

Polars	`df = df.with_columns([`
	`    pl.col('headline').str.strip_chars(),`
	`    pl.col('summary').fill_null("").cast(pl.Utf8).str.strip_chars()`
	`])`
	`df = df.filter(pl.col('headline') != "")`

4.2 Data Structure

a. Input Format:
CSV file from a merged data that has been scraped previously.

```
file_path = "nst_articles_final.csv"
```

```
df = pd.read_csv(file_path, encoding='latin1')
```

b. Output Format:
   Final structured dataset stored in CSV files after cleaning and transformation process.

```
df.to_csv("cleaned_pandas.csv", index=False, encoding="utf-8")
dask_df.to_csv("cleaned_dask.csv", index=False, encoding="utf-8")
polars_df.write_csv("cleaned_polars.csv")
```

   Fields (columns):
   1. Category
   2. Headline
   3. Summary
   4. Date
   5. Place
   6. Year
   7. Month

4.3 Transformation and Formatting

a. Clean and Parse Date Strings
   Dates are cleaned by removing “@”, trimming whitespace, and converting to datetime format.

Library	Code snippet

Pandas & Dask	`def parse_and_clean_dates(df,`
	`date_col='date'):`
	`    df[date_col] = pd.to_datetime(`
	`        df[date_col].str.replace('@', '',`
	`regex=False).str.strip(),`
	`        format='%b %d, %Y %I:%M%p',`
	`        errors='coerce'`
	`    )`
	`    return df`

Polars	`df = df.with_columns([`
	`        pl.col('date').str.replace('@', '',`
	`literal=False).str.strip_chars()`
	`    ])`
	
	`    df = df.with_columns([`
	
	`pl.col('date').str.strptime(pl.Datetime,`
	`format="%b %d, %Y %I:%M%p", strict=False)`
	`    ])`

b. Extract Place from Summary

Library	Code snippet

Pandas & Dask	`def extract_place(df, summary_col='summary'):`
	`    df['place'] =`
	`df[summary_col].str.extract(r'^([A-Z\s]+):')[0`
	`]`
	`    df['place'] =`
	`df['place'].str.strip().str.title()`
	`    df[summary_col] =`
	`df[summary_col].str.replace(r'^[A-Z\s]+:\s*',`

	`'', regex=True)`
	`    return df`

Polars	`# Extract place from summary using Polars string methods`
	`    place = df.select(`
	`pl.col('summary').str.extract(r'^([A-Z\s]+):')`
	`          .str.strip_chars()`
	`          .str.to_titlecase()`
	`    ).to_series()`
	
	`df = df.with_columns([place.alias('place')])`
	
	`    df = df.with_columns(`
	`df['summary'].str.replace(r'^[A-Z\s]+:\s*', '', literal=False).alias('summary')`
	`    )`

c. Format Category
Standardizes category values to title case

Library	Code snippet
Pandas & Dask	`df['category'] = df['category'].str.title()`

Polars	`# Capitalize first letter, lowercase rest for 'category'`
	`    df = df.with_columns(`
	`        (`
	`pl.col('category').str.to_lowercase()`
	`                        .str.slice(0,`

	`1).str.to_uppercase()` `+` `pl.col('category').str.slice(1, None)` `).alias('category')` `)`

d. Extract Year and Month

Library	Code snippet
Pandas & Dask	`df['year'] = df['date'].dt.year` `df['month'] = df['date'].dt.month`
Polars	`df = df.with_columns([` `pl.col('date').dt.year().alias('year'),` `pl.col('date').dt.month().alias('month'` `)` `])`

5.0 Optimization Techniques

In this project, three popular Python data processing libraries - Pandas, Dask, and Polars are implemented and compared with each other to evaluate their performance in cleaning and transforming the news article dataset. The goal was to identify the optimization techniques that improve processing time, CPU utilization, memory consumption, and overall throughput when handling big data.

5.1 Libraries Overview

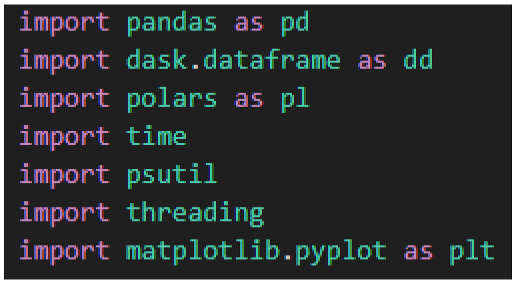
Library	Explanation
Pandas	Pandas is the traditional and widely-used library for data manipulation in Python. It offers an intuitive API and strong integration with the Python ecosystem. It processes data in a single-threaded manner, which can limit its performance on large datasets.
Dask	Dask extends Pandas by enabling parallel and distributed computing. It breaks data into smaller partitions and processes them concurrently across multiple CPU cores or even cluster nodes. Dask maintains a similar API to Pandas, allowing easier transition and scaling for larger datasets.
Polars	Polars is a modern DataFrame library built with Rust, designed for high-performance data processing. It uses native multithreading and SIMD optimizations to achieve significant speed-ups with lower memory overhead. It also provides expressive syntax for efficient data transformations.

5.2 Code Overview

This section describes the core implementation details and techniques used for data cleaning and optimization using Pandas, Dask and Polars.

1. Import Required Libraries

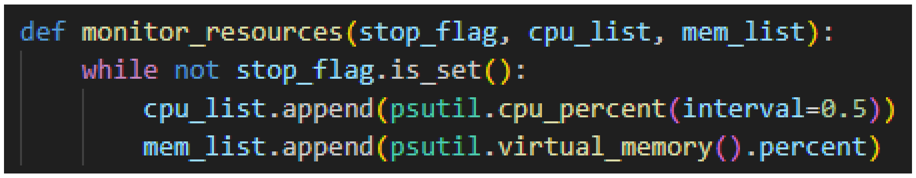
The project begins by importing essential libraries for data processing, resource monitoring, concurrency and visualization:



```python
import pandas as pd
import dask.dataframe as dd
import polars as pl
import time
import psutil
import threading
import matplotlib.pyplot as plt
```

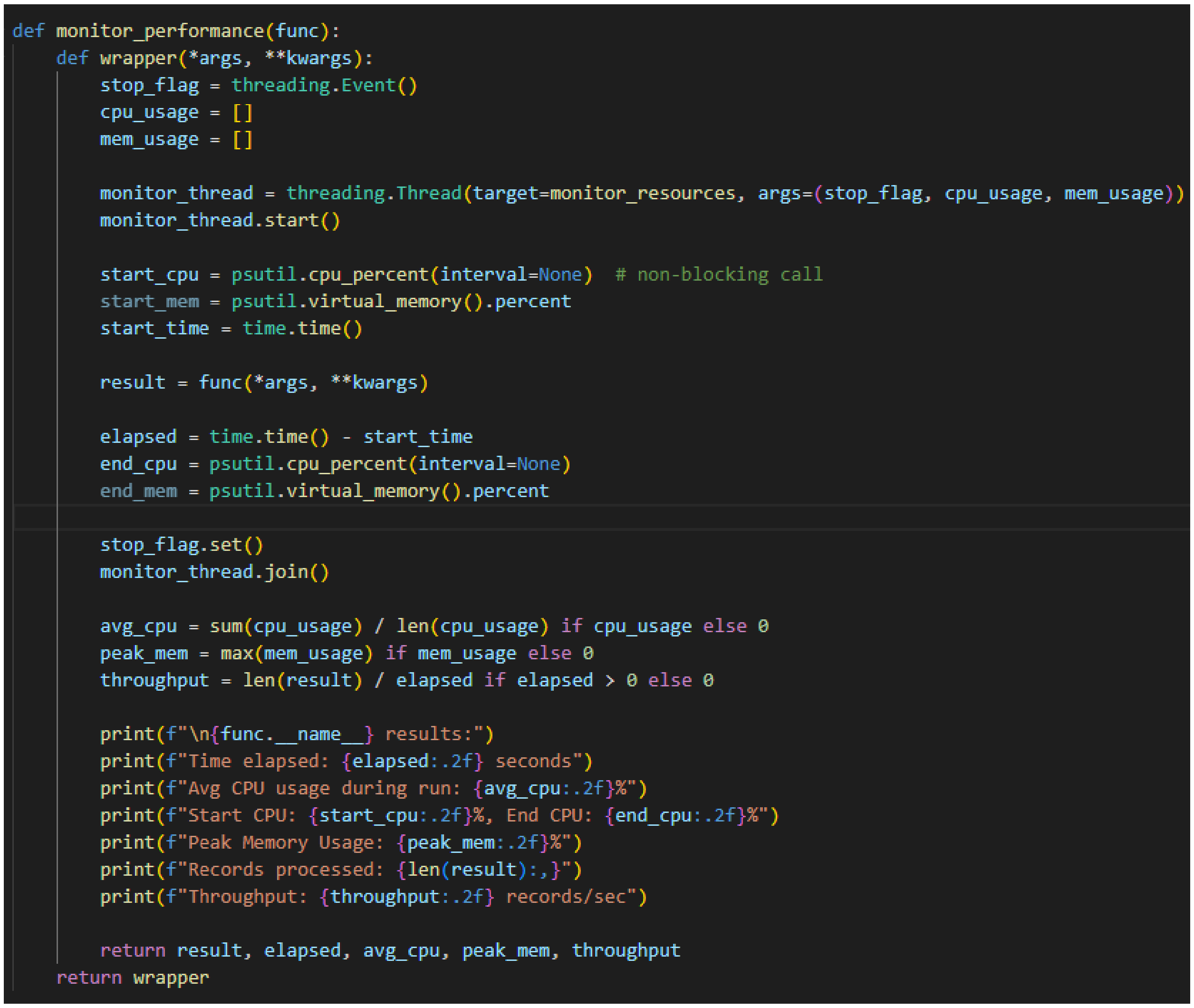
2. Resource Monitoring and Performance Measurement

A monitoring thread collects CPU and memory usage periodically to capture system resource usage during execution



```python
def monitor_resources(stop_flag, cpu_list, mem_list):
    while not stop_flag.is_set():
        cpu_list.append(psutil.cpu_percent(interval=0.5))
        mem_list.append(psutil.virtual_memory().percent)
```

A decorator named monitor_performance wraps the cleaning functions to measure the total execution time, average CPU usage, Peak memory usage, and also the throughput.

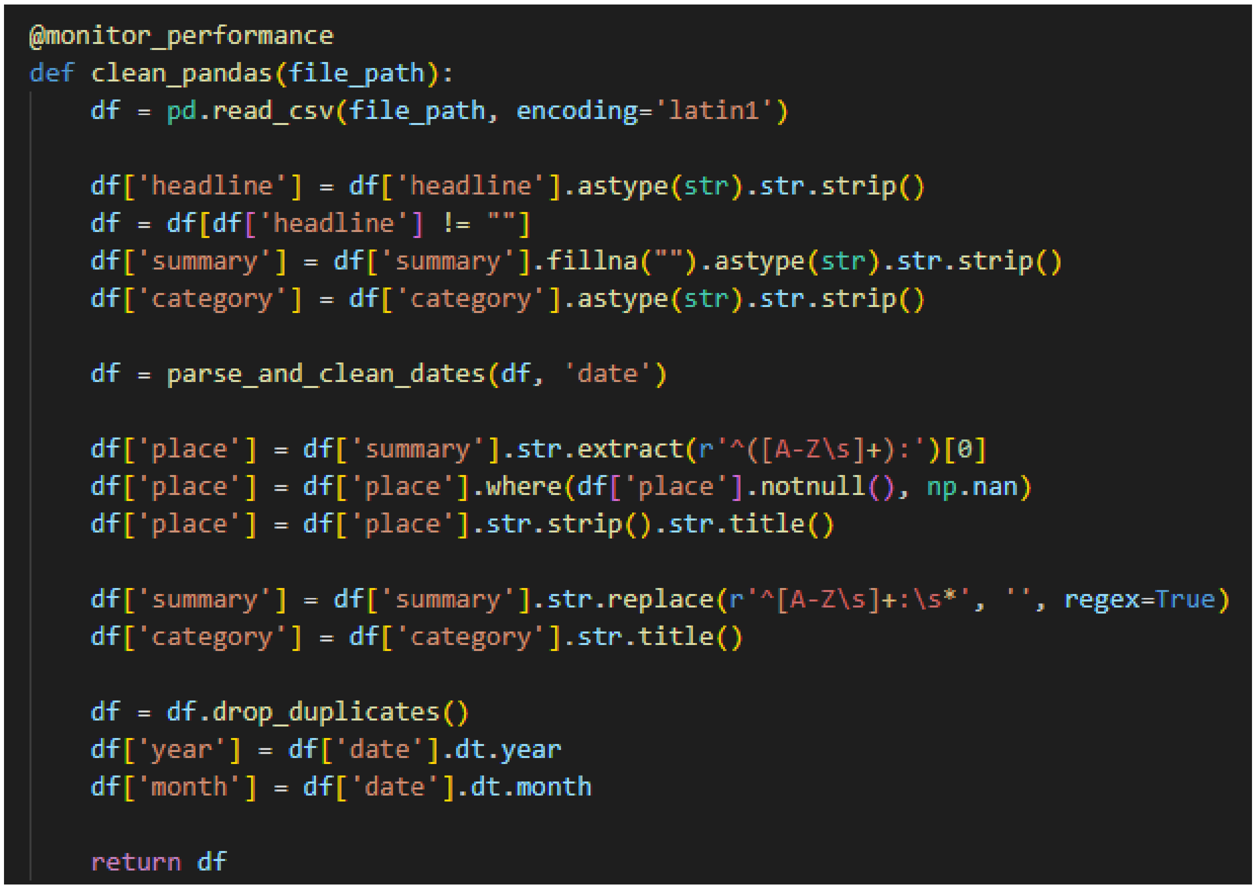


```python
def monitor_performance(func):
    def wrapper(*args, **kwargs):
        stop_flag = threading.Event()
        cpu_usage = []
        mem_usage = []

        monitor_thread = threading.Thread(target=monitor_resources, args=(stop_flag, cpu_usage, mem_usage))
        monitor_thread.start()

        start_cpu = psutil.cpu_percent(interval=None)  # non-blocking call
        start_mem = psutil.virtual_memory().percent
        start_time = time.time()

        result = func(*args, **kwargs)

        elapsed = time.time() - start_time
        end_cpu = psutil.cpu_percent(interval=None)
        end_mem = psutil.virtual_memory().percent

        stop_flag.set()
        monitor_thread.join()

        avg_cpu = sum(cpu_usage) / len(cpu_usage) if cpu_usage else 0
        peak_mem = max(mem_usage) if mem_usage else 0
        throughput = len(result) / elapsed if elapsed > 0 else 0

        print(f"\n{func.__name__} results:")
        print(f"Time elapsed: {elapsed:.2f} seconds")
        print(f"Avg CPU usage during run: {avg_cpu:.2f}%")
        print(f"Start CPU: {start_cpu:.2f}%, End CPU: {end_cpu:.2f}%")
        print(f"Peak Memory Usage: {peak_mem:.2f}%")
        print(f"Records processed: {len(result):,}")
        print(f"Throughput: {throughput:.2f} records/sec")

        return result, elapsed, avg_cpu, peak_mem, throughput
    return wrapper
```

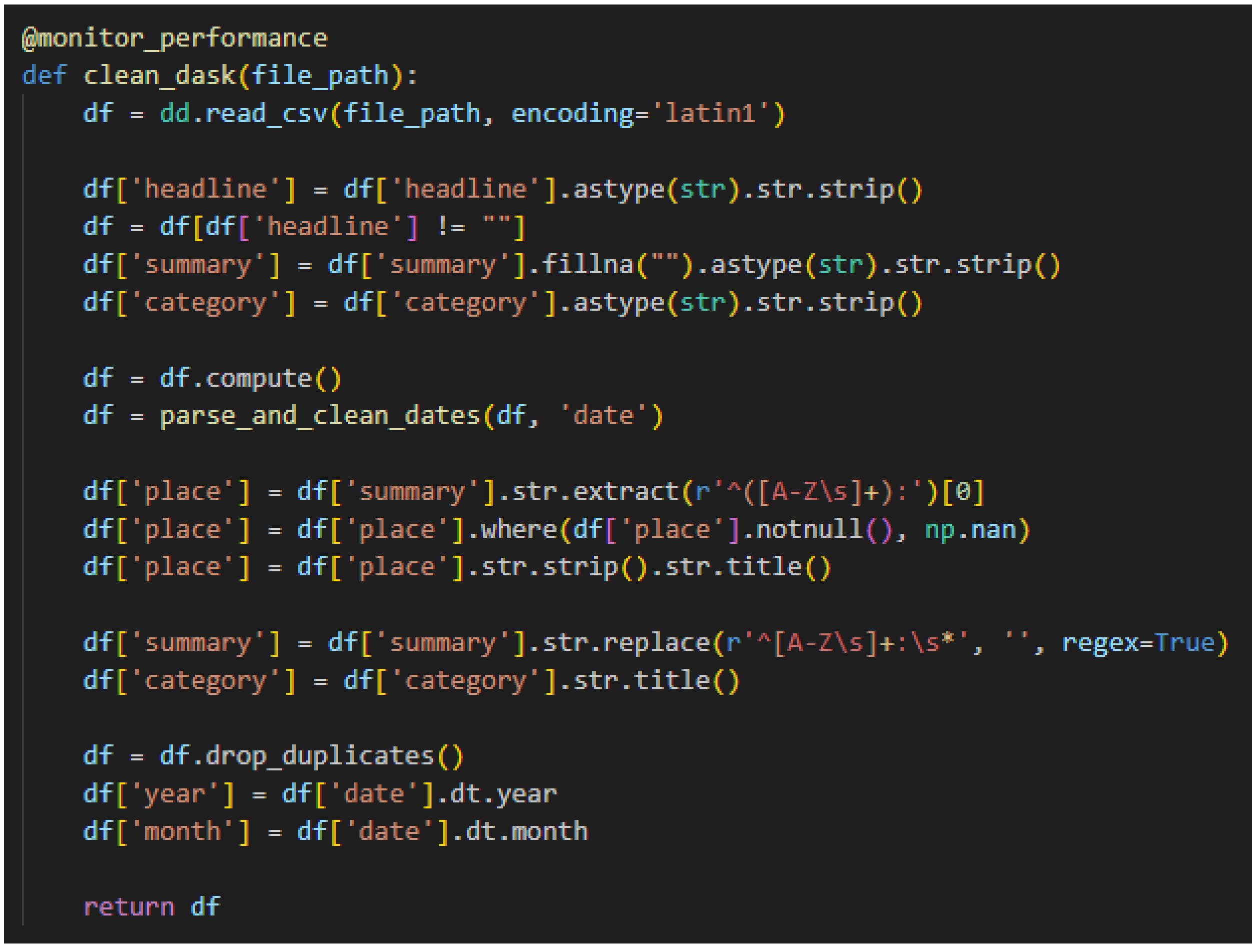
3. Data Cleaning Functions for Each Library

Pipelines that reads CSV and applies cleaning, data parsing, place extraction, deduplication, and feature engineering

a. Pandas Cleaning Pipeline
    i. Reads CSV with specified encoding
    ii. Strips white space and removes s headline
    iii. Cleans and standardizes text fields
    iv. Parses date strings into datetime objects, handling @ symbols
    v. Extracts places from the beginning of summary
    vi. Removes duplicates
    vii. Add year and month columns from date

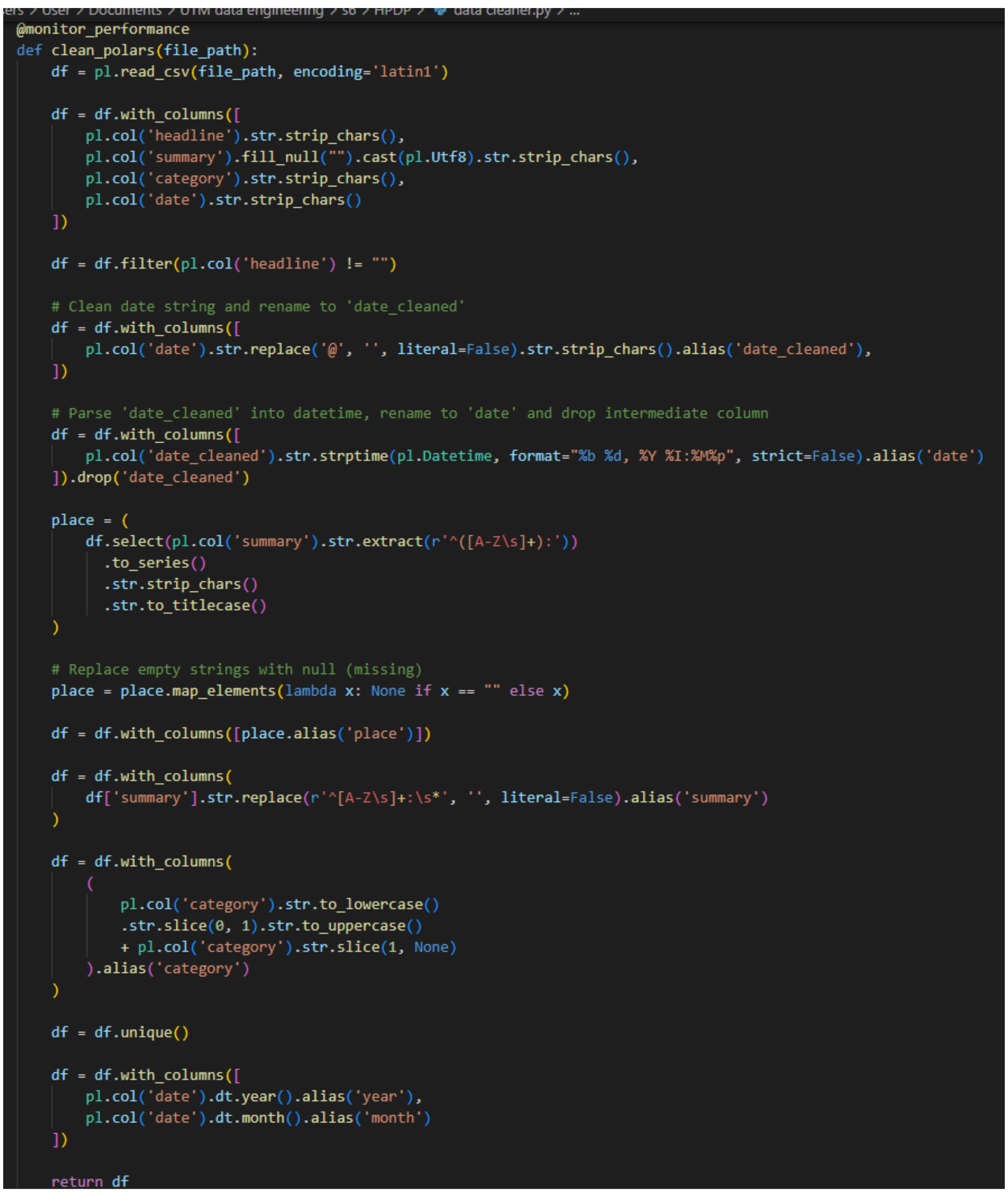

```python
@monitor_performance
def clean_pandas(file_path):
    df = pd.read_csv(file_path, encoding='latin1')

    df['headline'] = df['headline'].astype(str).str.strip()
    df = df[df['headline'] != ""]
    df['summary'] = df['summary'].fillna("").astype(str).str.strip()
    df['category'] = df['category'].astype(str).str.strip()

    df = parse_and_clean_dates(df, 'date')

    df['place'] = df['summary'].str.extract(r'^([A-Z\s]+):')[0]
    df['place'] = df['place'].where(df['place'].notnull(), np.nan)
    df['place'] = df['place'].str.strip().str.title()

    df['summary'] = df['summary'].str.replace(r'^[A-Z\s]+:\s*', '', regex=True)
    df['category'] = df['category'].str.title()

    df = df.drop_duplicates()
    df['year'] = df['date'].dt.year
    df['month'] = df['date'].dt.month

    return df
```

b. Dask Cleaning Pipeline
   i. Reads CSV as a Dask DataFrame to enable parallel reading
   ii. Applies similar cleaning logic as Pandas
   iii. Converts to Pandas DataFrame with .compute() for subsequent operations



```
@monitor_performance
def clean_dask(file_path):
    df = dd.read_csv(file_path, encoding='latin1')

    df['headline'] = df['headline'].astype(str).str.strip()
    df = df[df['headline'] != ""]
    df['summary'] = df['summary'].fillna("").astype(str).str.strip()
    df['category'] = df['category'].astype(str).str.strip()

    df = df.compute()
    df = parse_and_clean_dates(df, 'date')

    df['place'] = df['summary'].str.extract(r'^([A-Z\s]+):')[0]
    df['place'] = df['place'].where(df['place'].notnull(), np.nan)
    df['place'] = df['place'].str.strip().str.title()

    df['summary'] = df['summary'].str.replace(r'^[A-Z\s]+:\s*', '', regex=True)
    df['category'] = df['category'].str.title()

    df = df.drop_duplicates()
    df['year'] = df['date'].dt.year
    df['month'] = df['date'].dt.month

    return df
```

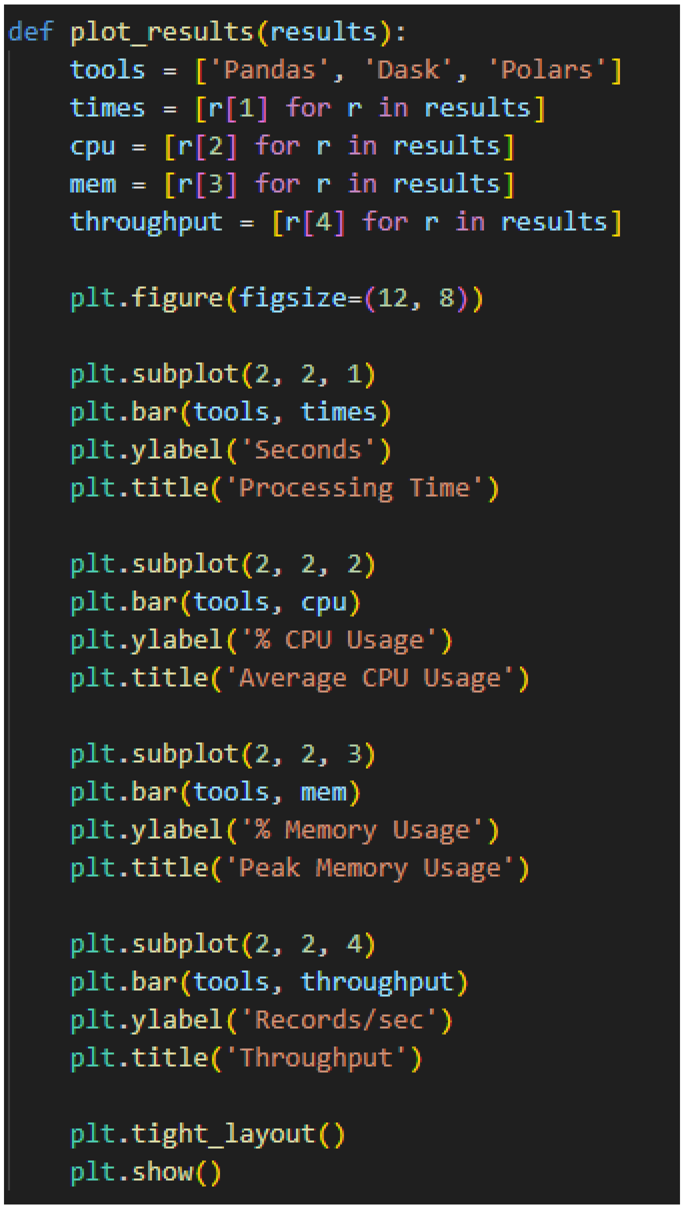
c. Polars Cleaning Pipeline
   i. Reads CSV with Polars
   ii. Uses Polars string function to clean text fields
   iii. Filters empty headlines
   iv. Cleans date strings, saves intermediate cleaned dates in date_cleaned to avoid duplicate column errors
   v. Parses date_cleaned into datetime, then drops the intermediate column
   vi. Extracts place from summary using regex and string operations
   vii. Removes place prefix from summary
   viii. Standardizes category capitalization
   ix. Removes duplicates
   x. Adds year and month columns



```
ers > User > Documents > UTM data engineering > s6 > HPDP > data cleaner.py > ...
@monitor_performance
def clean_polars(file_path):
    df = pl.read_csv(file_path, encoding='latin1')

    df = df.with_columns([
        pl.col('headline').str.strip_chars(),
        pl.col('summary').fill_null("").cast(pl.Utf8).str.strip_chars(),
        pl.col('category').str.strip_chars(),
        pl.col('date').str.strip_chars()
    ])

    df = df.filter(pl.col('headline') != "")

    # Clean date string and rename to 'date_cleaned'
    df = df.with_columns([
        pl.col('date').str.replace('@', '', literal=False).str.strip_chars().alias('date_cleaned'),
    ])

    # Parse 'date_cleaned' into datetime, rename to 'date' and drop intermediate column
    df = df.with_columns([
        pl.col('date_cleaned').str.strptime(pl.Datetime, format="%b %d, %Y %I:%M%p", strict=False).alias('date')
    ]).drop('date_cleaned')

    place = (
        df.select(pl.col('summary').str.extract(r'^([A-Z\s]+):'))
          .to_series()
          .str.strip_chars()
          .str.to_titlecase()
    )

    # Replace empty strings with null (missing)
    place = place.map_elements(lambda x: None if x == "" else x)

    df = df.with_columns([place.alias('place')])

    df = df.with_columns(
        df['summary'].str.replace(r'^[A-Z\s]+:\s*', '', literal=False).alias('summary')
    )

    df = df.with_columns(
        (
            pl.col('category').str.to_lowercase()
            .str.slice(0, 1).str.to_uppercase()
            + pl.col('category').str.slice(1, None)
        ).alias('category')
    )

    df = df.unique()

    df = df.with_columns([
        pl.col('date').dt.year().alias('year'),
        pl.col('date').dt.month().alias('month')
    ])

    return df
```

4. Visualization of Performance Metrics

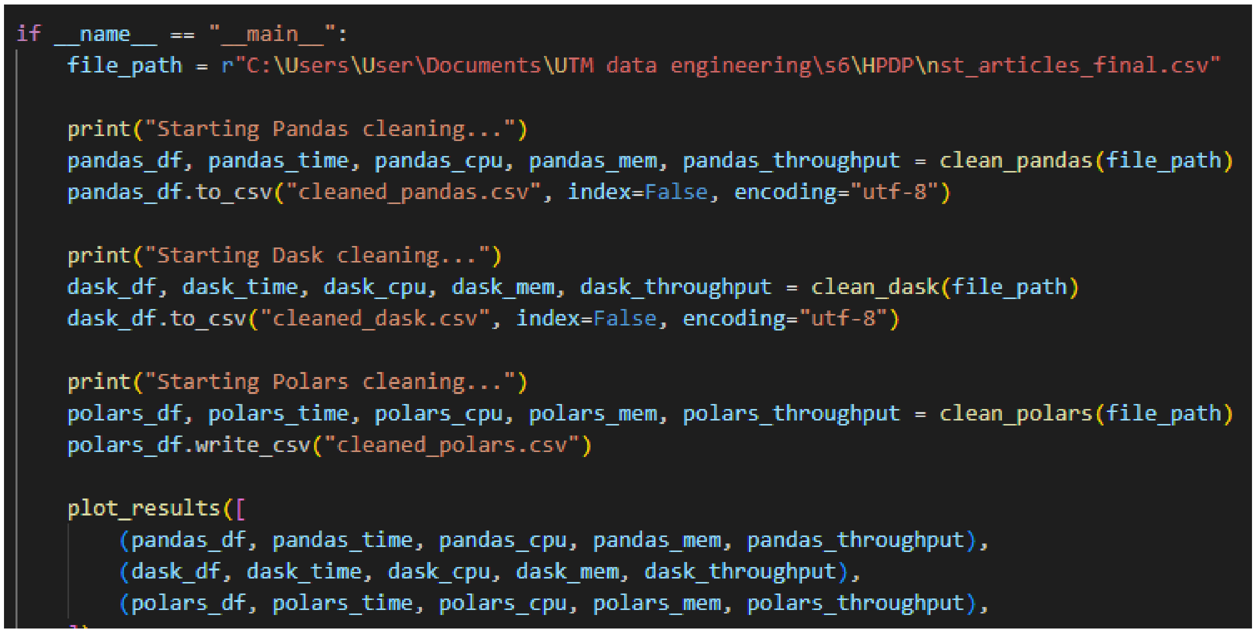
   A plotting function that displays bar charts comparing processing time, average CPU usage, peak memory usage, and throughput for the three libraries



```python
def plot_results(results):
    tools = ['Pandas', 'Dask', 'Polars']
    times = [r[1] for r in results]
    cpu = [r[2] for r in results]
    mem = [r[3] for r in results]
    throughput = [r[4] for r in results]

    plt.figure(figsize=(12, 8))

    plt.subplot(2, 2, 1)
    plt.bar(tools, times)
    plt.ylabel('Seconds')
    plt.title('Processing Time')

    plt.subplot(2, 2, 2)
    plt.bar(tools, cpu)
    plt.ylabel('% CPU Usage')
    plt.title('Average CPU Usage')

    plt.subplot(2, 2, 3)
    plt.bar(tools, mem)
    plt.ylabel('% Memory Usage')
    plt.title('Peak Memory Usage')

    plt.subplot(2, 2, 4)
    plt.bar(tools, throughput)
    plt.ylabel('Records/sec')
    plt.title('Throughput')

    plt.tight_layout()
    plt.show()
```

5. Execution Flow

   The main script that executes each cleaning function sequentially, saves cleaned data files, and visualizes the performance metrics



```python
if __name__ == "__main__":
    file_path = r"C:\Users\User\Documents\UTM data engineering\s6\HPDP\nst_articles_final.csv"

    print("Starting Pandas cleaning...")
    pandas_df, pandas_time, pandas_cpu, pandas_mem, pandas_throughput = clean_pandas(file_path)
    pandas_df.to_csv("cleaned_pandas.csv", index=False, encoding="utf-8")

    print("Starting Dask cleaning...")
    dask_df, dask_time, dask_cpu, dask_mem, dask_throughput = clean_dask(file_path)
    dask_df.to_csv("cleaned_dask.csv", index=False, encoding="utf-8")

    print("Starting Polars cleaning...")
    polars_df, polars_time, polars_cpu, polars_mem, polars_throughput = clean_polars(file_path)
    polars_df.write_csv("cleaned_polars.csv")

    plot_results([
        (pandas_df, pandas_time, pandas_cpu, pandas_mem, pandas_throughput),
        (dask_df, dask_time, dask_cpu, dask_mem, dask_throughput),
        (polars_df, polars_time, polars_cpu, polars_mem, polars_throughput),
```

6.0 Performance Evaluation

This section presents the evaluation of the data processing performance for the three libraries tested, which are Pandas, Dask, and Polars. The metrics used to analyze their performance include processing time, average CPU usage, peak memory consumption, and also throughput. These metrics provide insights into the efficiency of each approach when handling a large-scale news article dataset.

6.1 Summary of Results

Metric	Pandas	Dask	Polars
Time Elapsed (s)	1.30	1.56	0.19
Average CPU Usage (%)	31.03	27.37	28.90
Peak Memory Usage (%)	71.50	73.60	73.30
Throughput (record/sec)	77550	64766	529123
Records Processed	100962	100962	100962

6.2 Processing Time

Polars significantly outperformed Pandas and Dask in processing time, completing the cleaning task in just 0.19 seconds, which is approximately 7 times faster than Pandas and

8 times faster than Dask. This results shows that Polar's native multithreading and Rust-based optimizations enabled rapid execution on large datasets.

6.3 Average CPU Usage

Average CPU usage was comparable across the three libraries, ranging from about 27% to 31%. Pandas displayed the highest CPU utilization at 31%, followed by Polars' 29% and Dask's 27%. The high CPU utilization in Polars combined with its low processing time indicated its efficiency at parallel computation.

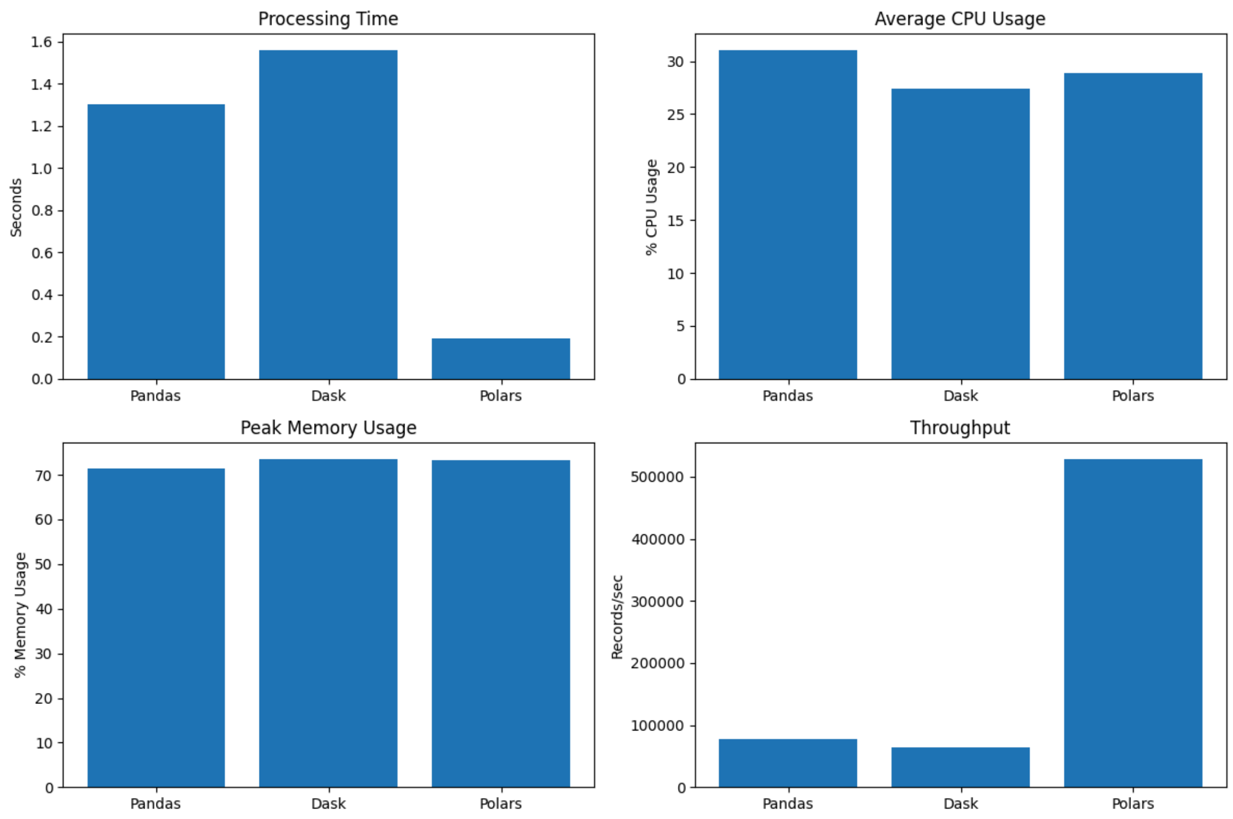
6.4 Peak Memory Usage

All three libraries exhibited similar peak memory usage between 71.6% and 73.6%. This indicates that memory consumption was not significantly impacted by the choice of the processing library. However, the slight increase in memory usage for Dask and Polars is most likely due to their parallel and multi-threaded processing architectures, which may hold additional intermediate data.

6.5 Throughput

Polars achieve an incredible throughput of over 529000 records per second, far surpassing both Pandas (~77500) and Das (~64700). This demonstrated Polars' capability to process large volumes of data very quickly, making it highly suitable for high-performance data processing pipelines.

6.6 Visualization



The performance evaluation shows that Polars excels in processing speed and throughput, making it the most efficient library for large-scale data cleaning tasks. Polars' native multithreading and Rust-based optimizations enable it to process over half a million records per second, significantly outpacing both Pandas and Dask. Although slower, Pandas maintains competitive CPU efficiency and slightly lower memory usage, benefiting from its mature and stable ecosystem, making it a reliable option for many data processing needs. Dask, designed for distributed and parallel computing, showed respectable performance but did not surpass Pandas or Polars in this test. This is likely due to the overhead in managing parallel tasks or because the dataset size comfortably fits into memory, limiting Dask's advantages. The accompanying visualization reinforces these findings by showing that all three libraries use CPU resources comparably and have similar peak memory usage, but Polars processes data much faster and with dramatically higher throughput. Overall, Polars offers a clear advantage in speed and efficiency, while Pandas and Dask provide balanced resource usage with their own strengths depending on the specific use case.

7.0 Challenges & Limitations

During the course of this project, our team had faced several challenges and limitations that impacted the data processing.

7.1 Target Website Selection and Scraping Challenges

The initial goal was to scrape data from PropertyGuru, due to its extensive property listings. However, the team faced significant challenges in scraping the website due to strong anti-scraping defenses such as IP blocking, dynamic content loading, CAPTCHA challenges, and frequent changes to site structure. These obstacles made automated scraping unreliable. In the end our team compromised and shifted our focus to the New Straits Times (NST) website, as it is more accessible.

7.2 Tooling and Environment Constraints

Our team initially planned to use Scrapy in the Google Colab environment. However, compatibility issues and resource limitations when processing large datasets caused us to switch. The team transitioned to Selenium, running locally on Visual Studio Code, to better handle dynamic content and site interactions. While Selenium provided greater control, it required more setup and management of browser automation compared to Scrapy.

7.3 Data Cleaning And Transformation Challenge

Another challenge faced by our team during the project was trying to maintain consistency and accuracy while performing data cleaning and transformation across different processing libraries. Each library has its own syntax, behaviours and limitations, making it hard to create a standardized workflow. As an example, certain string operations and date time manipulator that were straightforward in Pandas required an alternative method or workaround in Dask and Polars. These inconsistencies increased development time and required careful validation to ensure data integrity across all frameworks.

8.0 Conclusion & Future Work

Throughout this project, we successfully designed and implemented a high-performance multithreaded web scraping system that was capable of extracting over 100,000 news articles from the New Straits Times (NST) website. By combining Selenium with Python's threading module, we were able to navigate dynamic content efficiently and store the results in a structured format suitable for analysis.

For the data processing phase, we explored and compared three popular libraries—Pandas, Dask, and Polars. Our experiments showed that while Pandas remains dependable and user-friendly for smaller datasets, Polars clearly stood out in terms of speed and throughput, processing over 500,000 records per second. Dask, although built for scalability, showed less benefit in our context due to the dataset fitting into local memory.

We also faced several challenges, from changing our target website due to scraping restrictions, to resolving environment compatibility issues that impacted our initial plan. Despite these setbacks, the team was able to adapt and deliver a working, optimized solution with meaningful performance evaluation.

Looking ahead, we plan to enhance this system by integrating distributed scraping techniques, using tools like Playwright or headless Chromium clusters. Additionally, we aim to store the data in a database rather than CSV files and explore real-time scraping or stream processing approaches. These improvements would allow our system to scale further and adapt to more demanding use cases.

9.0 References

1. McKinney, W. (2012). Python for Data Analysis. O'Reilly Media.
2. Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. Proceedings of the 14th Python in Science Conference.
3. Polars User Guide – https://pola-rs.github.io/polars-book/
4. Selenium Documentation – https://www.selenium.dev/documentation/
5. Python Official Docs – https://docs.python.org/3/
6. New Straits Times – https://www.nst.com.my/
7. Web Scraping Best Practices – https://developers.google.com/search/docs/crawling-indexing/scrape
8. Psutil Documentation – https://psutil.readthedocs.io/en/latest/

10.0 Appendices

10.1 Web Scraper Source Code

Code snippet

```python
import csv
import time
from queue import Queue
import threading  # ✅ Optimization: Enables multithreading
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
from webdriver_manager.chrome import ChromeDriverManager
from selenium.common.exceptions import TimeoutException,
NoSuchElementException, WebDriverException
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC


def create_driver():
    options = Options()

    # ✅ Optimization: Use headless mode and disable unnecessary
resources for faster loading
    options.add_argument("--headless")
    options.add_argument("--disable-gpu")
    options.add_argument("--no-sandbox")
    options.add_argument("--ignore-certificate-errors")
    options.add_argument("--window-size=1920,1080")

    # ✅ Optimization: Block images, stylesheets, and fonts to speed up
page load
    options.add_experimental_option("prefs", {
        "profile.managed_default_content_settings.images": 2,
        "profile.managed_default_content_settings.stylesheets": 2,
        "profile.managed_default_content_settings.fonts": 2
    })

    options.add_argument("user-agent=Mozilla/5.0 ... Safari/537.36")

    return
webdriver.Chrome(service=Service(ChromeDriverManager().install()),
options=options)


def scrape_with_driver(driver, page_num):
    url = f"https://www.nst.com.my/business?page={page_num}"
    results = []

    try:
        driver.get(url)
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CSS_SELECTOR,
".article-teaser"))
        )

        articles = driver.find_elements(By.CSS_SELECTOR,
".article-teaser")

        for article in articles:
            try:
                # ✅ Data Cleaning: Remove whitespace and validate
non-empty headline
                headline = article.find_element(By.CSS_SELECTOR,
".field-title").text.strip()
                if not headline:
                    continue

                category = article.find_element(By.CSS_SELECTOR,
".field-category").text.strip()
                date = article.find_element(By.CSS_SELECTOR,
".created-ago").text.strip()

                # ✅ Data Cleaning: Extract summary text only from
specific element
```

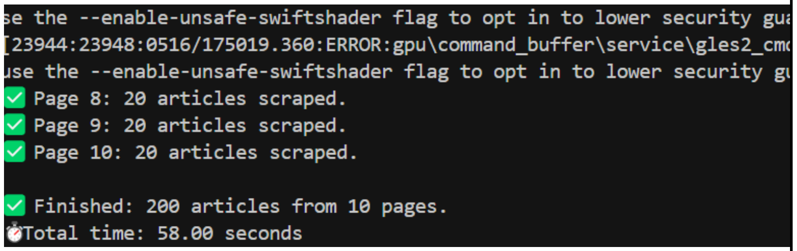
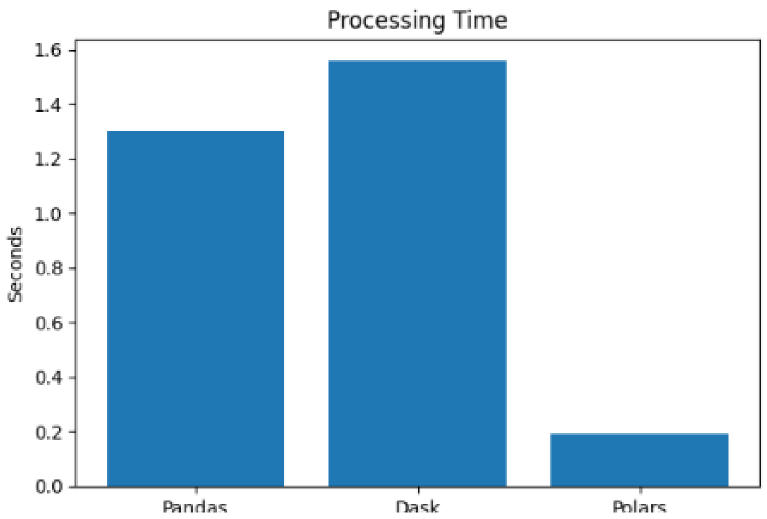
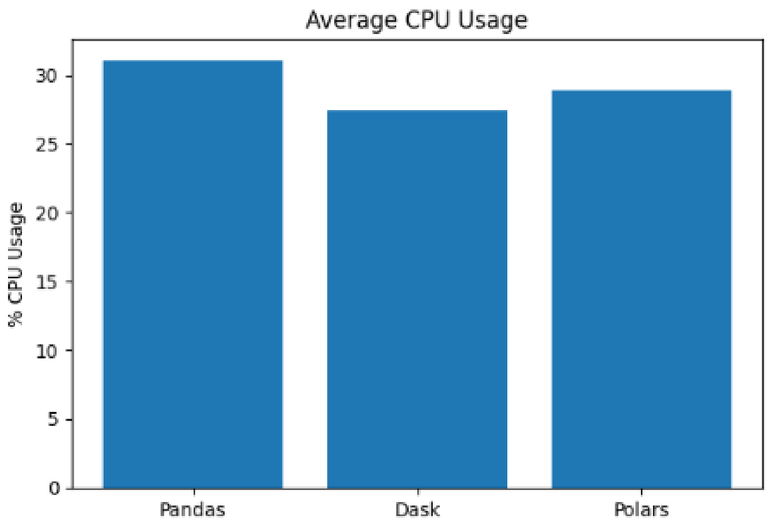
```
                try:
                    summary_elem = article.find_element(By.CSS_SELECTOR,
".field-body")
                    summary = summary_elem.text.strip()
                except NoSuchElementException:
                    summary = ""  # ✅ Data Cleaning: Fallback if
summary is missing

                results.append({
                    "category": category,
                    "headline": headline,
                    "summary": summary,
                    "date": date
                })

            except NoSuchElementException:
                continue

        print(f"✅ Page {page_num}: {len(results)} articles scraped.")
    except Exception as e:
        print(f"❌ Error on page {page_num}: {e}")
    return results


def worker(queue, output_list):
    driver = create_driver()
    while not queue.empty():
        page = queue.get()
        result = scrape_with_driver(driver, page)
        output_list.extend(result)
        queue.task_done()
    driver.quit()


def main():
    total_pages = 10
    num_threads = 4
    q = Queue()
    results = []
    # ---------- START TIMER ----------
    start_time = time.time()
    # ---------------------------------

    # ✅ Optimization: Queue used to distribute workload across threads
    for page in range(1, total_pages + 1):
        q.put(page)

    threads = []
    for _ in range(num_threads):
        # ✅ Optimization: Multithreading implementation to speed up
scraping
        t = threading.Thread(target=worker, args=(q, results))
        t.start()
        threads.append(t)

    for t in threads:
        t.join()

    # ✅ Data Output: Save the cleaned and structured data to CSV
    with open("nst_articles_optimized.csv", "w", newline="",
encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=["category", "headline",
"summary", "date"])
        writer.writeheader()
        writer.writerows(results)

    # ---------- END TIMER & REPORT ----------
    elapsed = time.time() - start_time
    print(f"\n✅ Finished: {len(results)} articles from {total_pages}
pages.")
    print(f"⏱️ Total time: {elapsed:.2f} seconds")
    # ----------------------------------------


if __name__ == "__main__":
    main()
```

10.2 Performance Evaluation Charts

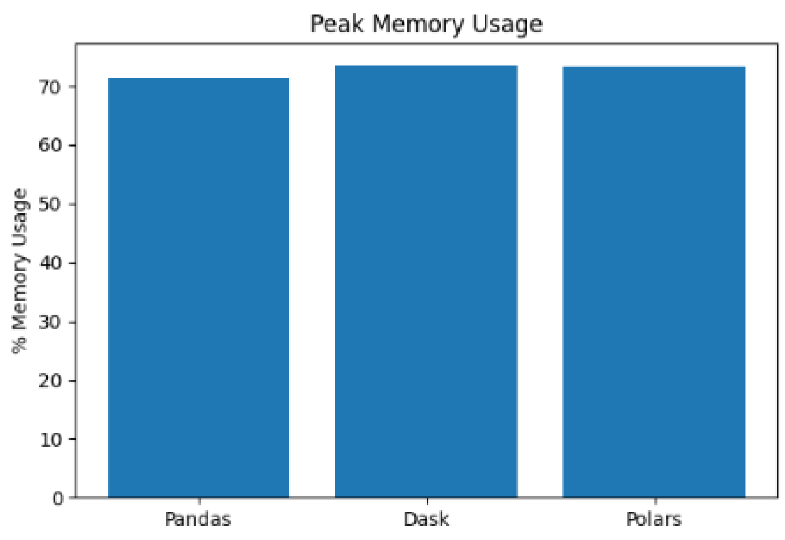
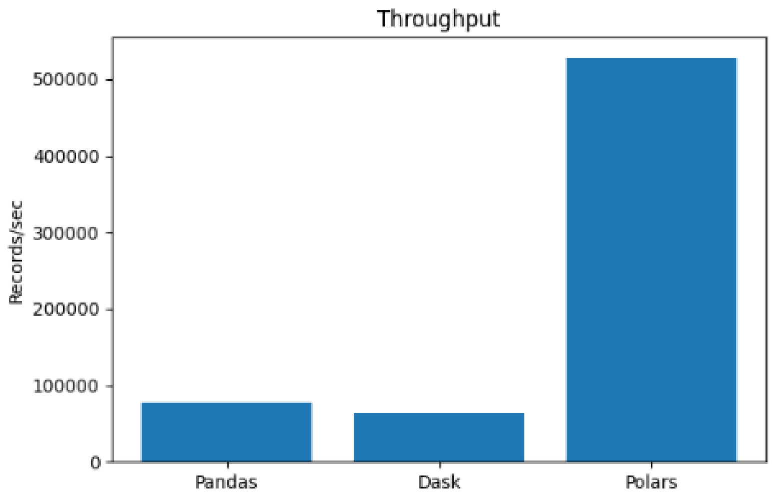
Code snippet

```
def plot_results(results):
    tools = ['Pandas', 'Dask', 'Polars']
    times = [r[1] for r in results]
    cpu = [r[2] for r in results]
    mem = [r[3] for r in results]
    throughput = [r[4] for r in results]

    plt.figure(figsize=(12, 8))

    plt.subplot(2, 2, 1)
    plt.bar(tools, times)
    plt.ylabel('Seconds')
    plt.title('Processing Time')

    plt.subplot(2, 2, 2)
    plt.bar(tools, cpu)
    plt.ylabel('% CPU Usage')
    plt.title('Average CPU Usage')

    plt.subplot(2, 2, 3)
    plt.bar(tools, mem)
    plt.ylabel('% Memory Usage')
    plt.title('Peak Memory Usage')

    plt.subplot(2, 2, 4)
    plt.bar(tools, throughput)
    plt.ylabel('Records/sec')
    plt.title('Throughput')

    plt.tight_layout()
    plt.show()
```

10.3 Output Screenshots

Description	Output
Web scraper output	
Performance evaluation chart 1	
Performance evaluation chart 2	

```
se the --enable-unsafe-swiftshader flag to opt in to lower security gua
[23944:23948:0516/175019.360:ERROR:gpu\command_buffer\service\gles2_cmd
use the --enable-unsafe-swiftshader flag to opt in to lower security gu
✅ Page 8: 20 articles scraped.
✅ Page 9: 20 articles scraped.
✅ Page 10: 20 articles scraped.

✅ Finished: 200 articles from 10 pages.
⏱️Total time: 58.00 seconds
```

Performance evaluation chart 3	
Performance evaluation chart 4	 Throughput Records/sec 0 100000 200000 300000 400000 500000 Pandas Dask Polars