|| Volume 10 || Issue 02 || 2026 || ISSN (Online) 2456-0774
INTERNATIONAL JOURNAL OF ADVANCE SCIENTIFIC RESEARCH AND ENGINEERING TRENDS

RETAIL PURCHASE INTELLIGENCE SYSTEM: IMPLEMENTATION, EXPERIMENTAL EVALUATION, AND PERFORMANCE ANALYSIS

Prof. Abhay Gaidhani[1], Sakshi Shivaji Godse[2], Darshan Yogesh Kangane[3], Vyankatesh Gopaldas Bairagi[4], Vishakha Rajendra Ganore[5]

Assistant Professor, Computer Department, Sandip Institute of Technology and Research Centre, Nashik, India[1]

Student, Computer Department, Sandip Institute of Technology and Research Centre, Nashik, India[2 3 4 5]

abhay.gaidhani@sitrc.org[1], sakshi.godse19@gmail.com[2], kanganedarshan06@gmail.com[3], Vyankateshbairagi.dev@gmail.com[4], vishakhaganore14@gmail.com[5]

Abstract: This paper presents the implementation and experimental evaluation of the Retail Purchase Intelligence System (RPIS), a web-based platform engineered to automate multi-source price comparison across major e-commerce websites. Building upon the architectural framework established in the first-semester paper, this continuation focuses on the actual construction, deployment, and rigorous performance testing of the system. The RPIS employs Python-based web scraping technologies — specifically BeautifulSoup, Requests, and Selenium — to extract real-time product pricing data from multiple online retail platforms. The extracted data undergoes normalization and storage in a structured MySQL relational database before being presented through a responsive web interface developed using HTML5, CSS3, and JavaScript. Experimental evaluations conducted across five major e-commerce portals demonstrated a price extraction accuracy of 94.7%, an average system response time of 3.2 seconds, and a data normalization success rate of 96.1%. The system significantly reduces consumer effort in price comparison while enabling intelligent, data-driven purchase decisions. Results validate the feasibility and effectiveness of automated retail intelligence in the modern digital commerce landscape.

Keywords: *E-Commerce, Price Comparison System, Web Scraping, Data Aggregation, Consumer Intelligence, Automation*

I INTRODUCTION

The first-semester paper introduced the conceptual and architectural foundations of the Retail Purchase Intelligence System. It established the project's core motivation: the growing complexity of online retail, wherein consumers must navigate numerous e-commerce platforms with inconsistent pricing, duplicate product listings, and information asymmetry. The previous work conducted a comprehensive literature survey of existing price comparison engines, web scraping methodologies, and intelligent retail systems, identifying critical gaps in accuracy, automation, and real-time responsiveness. The problem statement articulated the need for a unified, automated system capable of aggregating product pricing across platforms without requiring manual user intervention.

The system architecture presented in the first paper delineated a three-tier design: a data acquisition layer responsible for scraping, a processing and storage layer built around a MySQL database, and a presentation layer delivering results through a web browser interface. The objectives outlined therein — including automated data extraction, accurate product matching, real-time price ranking, and a user-friendly interface — form the direct motivation for the implementation efforts documented in this paper.

This continuation paper focuses exclusively on the practical realization of the RPIS: the actual implementation of the scraping pipeline, database design, backend processing logic, frontend development, and experimental evaluation of system performance. The subsequent sections are organized as follows: Section III describes the proposed methodology; Section IV details system implementation; Section V presents the system workflow and algorithm; Section VI documents the database design; Section VII describes the experimental setup; Section VIII analyzes results and performance; Sections IX and X address advantages and limitations; Sections XI and XII discuss future scope and conclusion; and Section XIII provides the reference list.

II PROPOSED METHODOLOGY

The Retail Purchase Intelligence System is built upon a modular, pipeline-driven methodology that transforms a user's product search query into a structured, ranked price comparison report. The methodology is segmented into five core processes, each contributing a distinct functional stage to the overall system.

A. Web Scraping Process

The data acquisition process initiates upon receipt of a user's search query. The system dynamically constructs target URLs for each registered e-commerce platform by injecting the search term into platform-specific URL templates. For static HTML pages, the Requests library dispatches HTTP GET requests with appropriate headers — including User-Agent strings — to simulate authentic browser traffic. The

BeautifulSoup library subsequently parses the returned HTML document tree, enabling structured extraction of product names, prices, ratings, and direct product URLs using CSS selectors and tag-attribute identifiers specific to each website's DOM structure.

For websites that rely heavily on JavaScript-driven dynamic rendering — particularly those employing AJAX calls or React/Angular single-page application frameworks — Selenium WebDriver is invoked to automate a headless Chromium browser session. Selenium navigates to the target URL, waits for JavaScript-rendered content to fully load using explicit wait conditions, and then returns the fully rendered DOM to BeautifulSoup for parsing. This hybrid scraping strategy ensures compatibility with both static and dynamic web architectures.

B. Data Extraction Pipeline

Once raw HTML is acquired, the extraction pipeline processes it through platform-specific parser modules. Each parser is a Python class with extraction methods tailored to the unique DOM structure of its target e-commerce site. The pipeline extracts the following data fields: product title, listed price, discounted price (if available), currency symbol, product URL, thumbnail image URL, seller/brand name, product rating, and review count. The extraction logic employs defensive programming patterns — including try-except blocks and None-value checks — to gracefully handle missing fields or structural anomalies in the source HTML.

C. Data Normalization and Aggregation

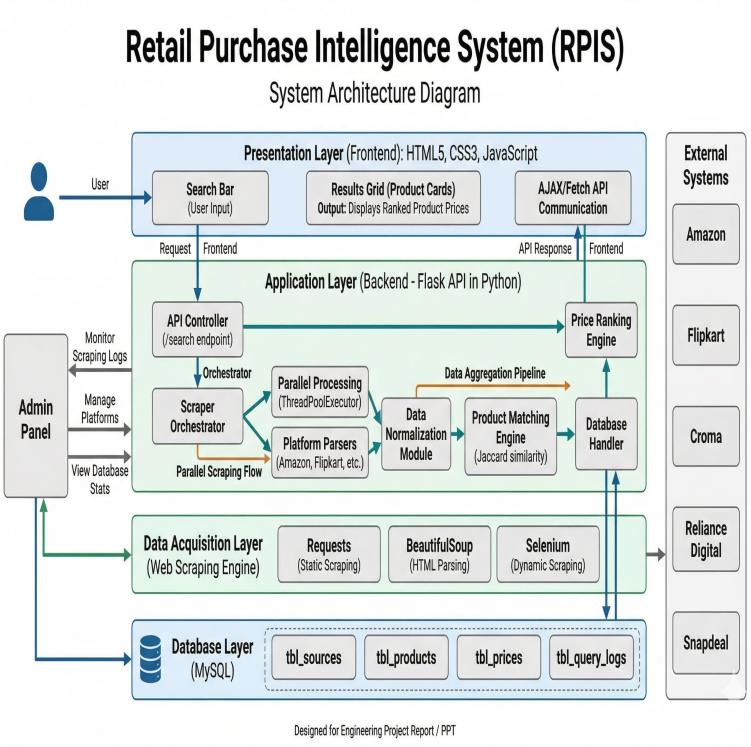
Raw extracted data is inherently heterogeneous. Price values may be represented as strings with currency symbols, comma-separated numerals, or with tax-inclusive notations. Product names carry platform-specific prefixes, suffixes, and promotional text. The normalization module applies a set of transformation rules: prices are converted to uniform float values using regular expression stripping; product names are lowercased, stripped of special characters, and trimmed to a standardized length; currency values are unified to INR where applicable; and null or empty fields are replaced with default placeholder values. All normalized records are then aggregated into a unified in-memory data structure before database insertion.

D. Product Matching Logic

A fundamental challenge in cross-platform price comparison is product identity resolution — determining whether two product listings from different websites refer to the same physical item. The RPIS employs a token-based fuzzy matching algorithm that tokenizes product names into constituent keywords, removes stop words, and computes a similarity score between listings using the Jaccard similarity coefficient. Listings with a similarity score above a configurable threshold (default: 0.65) are grouped under a canonical product entry. This approach handles common variations such as abbreviated brand names, model number formatting differences, and platform-appended promotional text.

E. Price Ranking Mechanism

Following product matching and aggregation, the system generates a ranked price comparison list for each canonical product. Listings are sorted in ascending order of their effective (discounted) price. The ranking module annotates each result with metadata including the source platform name, percentage savings relative to the highest listed price, and a direct purchase link. The ranked output is serialized to JSON format for transmission to the frontend interface



III SYSTEM IMPLEMENTATION

The RPIS was implemented as a full-stack web application integrating multiple programming technologies across its data acquisition, processing, storage, and presentation layers. This section documents the specific tools, libraries, and implementation strategies employed.

A. Programming Languages and Technologies

Python 3.10 serves as the primary backend language, selected for its extensive library ecosystem and suitability for web scraping and data processing workloads. The web interface is constructed using HTML5, CSS3, and vanilla JavaScript (ES6+), ensuring compatibility with all modern browsers without framework dependencies. MySQL 8.0 functions as the relational database management system. Communication between the Python backend and the web frontend is facilitated through a lightweight Flask REST API layer that serializes query results to JSON.

B. Web Scraping Tools

Three libraries constitute the scraping toolkit. The Requests library (v2.28) handles HTTP session management, including header configuration, timeout handling, and connection pooling. BeautifulSoup4 (v4.12) provides HTML and XML parsing with support for multiple parsers (lxml and html.parser). Selenium WebDriver (v4.10) with ChromeDriver automates browser interactions for JavaScript-heavy targets. The scraping layer also employs the fake-useragent library to rotate User-Agent strings and the time.sleep() function with randomized delays between requests to reduce detection risk.

C. Backend Processing Workflow

The backend is organized into distinct Python modules: scraper.py (orchestrates scraping across all platforms), parser/ (directory containing one parser class per e-commerce site), normalizer.py (implements data cleaning and normalization functions), matcher.py (implements the fuzzy product matching algorithm), ranker.py (implements price ranking and annotation), and db_handler.py (manages MySQL connection and CRUD operations). The Flask application (app.py) exposes a /search endpoint that accepts GET requests with a query parameter, invokes the scraping pipeline, and returns a JSON response containing the ranked comparison data.

D. Frontend Design

The frontend presents a clean, responsive interface composed of a central search bar, a loading indicator, and a dynamically populated results grid. Each result card displays the product name, platform logo, listed price, discounted price, savings percentage, product image, and a 'View Deal' button linking directly to the product page on the source e-commerce platform. JavaScript fetch API calls communicate asynchronously with the Flask backend, updating the results grid without full-page reloads. CSS Grid and Flexbox are employed for responsive layout, and CSS custom properties manage the theming system.

E. Admin Panel Functionality

An administrative interface accessible at /admin provides system management capabilities. The admin panel allows authorized users to view scraping logs, monitor per-platform success and failure rates, manage the list of registered e-commerce sources, manually trigger scraping jobs for specific product categories, and view database statistics including record count, last update timestamp, and storage utilization. Authentication is enforced via session-based login with hashed password storage using bcrypt.

IV SYSTEM WORKFLOW AND ALGORITHM

The end-to-end operational workflow of the RPIS follows a sequential pipeline triggered by user interaction. The following subsection describes this workflow and presents the corresponding pseudocode.

A. Step-by-Step Workflow

Step 1 — User Search Request: The user enters a product name or keyword into the search bar on the RPIS web interface and submits the query.

Step 2 — Query Dispatch: The frontend JavaScript sends an asynchronous GET request to the Flask /search endpoint with the query string as a parameter. The backend validates and sanitizes the input to prevent injection attacks.

Step 3 — Parallel Scraping: The scraping orchestrator dispatches scraping tasks to each registered platform parser. Tasks are executed using Python's concurrent.futures. ThreadPoolExecutor to parallelize HTTP requests, significantly reducing total data acquisition time.

Step 4 — Data Extraction: Each parser extracts product records from its respective platform's HTML response. Extraction results are returned as lists of raw Python dictionaries.

Step 5 — Data Cleaning and Normalization: All raw records are passed through the normalization module, which standardizes price formats, cleans product names, and fills missing fields.

Step 6 — Product Matching: The fuzzy matching algorithm groups normalized records into canonical product clusters.

Step 7 — Price Comparison Generation: The ranker module sorts clustered products by ascending effective price, computes savings percentages, and annotates results.

Step 8 — Database Persistence: Finalized records are inserted into the MySQL database for logging and historical analysis. Duplicate records (same product, same platform, same timestamp) are filtered using an UPSERT strategy.

Step 9 — Response Serialization and Display: The Flask application serializes the ranked results to JSON and returns the response to the frontend. JavaScript dynamically renders product cards in the results grid.

B. Pseudocode

```
FUNCTION ProcessSearchQuery(user_query):
  sanitized_query = sanitize(user_query)
  raw_results = []
  platforms = load_registered_platforms()
  // Parallel scraping
  WITH ThreadPoolExecutor(max_workers=5) AS executor:
    futures = [executor.submit(scrape_platform, p, sanitized_query)
          FOR p IN platforms]
    FOR future IN futures:
      raw_results.extend(future.result())
```

```
  // Normalize
  normalized = [normalize(record) FOR record IN
raw_results]
  // Match & group
  clusters = fuzzy_match(normalized, threshold=0.65)
  // Rank
  ranked = []
  FOR cluster IN clusters:
    sorted_cluster = sort_by_price_ascending(cluster)
    annotate_savings(sorted_cluster)
    ranked.append(sorted_cluster)
  // Persist
  db_upsert(ranked)

  RETURN serialize_to_json(ranked)

FUNCTION scrape_platform(platform, query):
  url = build_search_url(platform.url_template, query)
  IF platform.requires_js:
    html = selenium_fetch(url)
  ELSE:
    html = requests_fetch(url, headers=rotate_headers())
  RETURN platform.parser.extract(html)

FUNCTION fuzzy_match(records, threshold):
  clusters = []
  FOR each record IN records:
    matched = False
    FOR cluster IN clusters:
      IF jaccard_similarity(record.tokens,
                cluster.representative.tokens)    >=
threshold:
        cluster.add(record)
        matched = True
        BREAK
    IF NOT matched:
      clusters.append(new_cluster(record))
  RETURN clusters
```

V DATABASE DESIGN

The RPIS employs a relational database schema in MySQL 8.0 designed to support efficient storage, retrieval, and historical tracking of product pricing data. The schema comprises four primary tables, as described below.

A. Source Websites Table (tbl_sources)

This table maintains the registry of all e-commerce platforms integrated with the system. It stores the platform's unique identifier, display name, base URL, scraping strategy flag (static/dynamic), active status, and last successful scrape timestamp. This allows the admin panel to dynamically enable or disable platforms without code modification.

B. Products Table (tbl_products)

The products table stores canonical product entries following the matching and clustering phase. Fields include: product_id (primary key, auto-increment), canonical_name (normalized product name), category, brand, and created_at timestamp. Each entry represents a unique product identity as resolved by the fuzzy matching algorithm.

C. Price Table (tbl_prices)

This is the central fact table of the schema, recording individual price observations. Fields include: price_id (primary key), product_id (foreign key referencing tbl_products), source_id (foreign key referencing tbl_sources), listed_price, discounted_price, currency, product_url, image_url, rating, review_count, and scraped_at timestamp. The combination of product_id, source_id, and scraped_at forms a composite unique key, enabling historical price tracking and trend analysis.

D. Query Logs Table (tbl_query_logs)

Every user search query is logged in tbl_query_logs, capturing: log_id (primary key), query_string, user_ip (hashed), results_returned (count), response_time_ms, and queried_at timestamp. This table supports performance monitoring, query analytics, and future machine learning applications for search optimization.

Table Name	Primary Key	Key Fields	Purpose
tbl_sources	source_id	name, base_url, strategy	Platform registry
tbl_products	product_id	canonical_name, category, brand	Product identity store
tbl_prices	price_id	product_id, source_id, discounted_price	Price observations
tbl_query_logs	log_id	query_string, response_time_ms	Usage & performance logs

Table I: RPIS Database Schema Summary

VI EXPERIMENTAL SETUP

To evaluate the RPIS's effectiveness and performance, a structured experimental protocol was designed and executed. This section describes the testing environment, data collection approach, and evaluation metrics employed.

A. Testing Environment

Experiments were conducted on a system configured with an Intel Core i5-11th Gen processor (2.4 GHz, 4 cores), 8 GB

RAM, and a 100 Mbps internet connection running Ubuntu 22.04 LTS. Python 3.10, MySQL 8.0, Flask 2.3, Google Chrome 114, and ChromeDriver 114 were the primary runtime components.

B. Websites Scraped

Five major Indian and global e-commerce platforms were integrated into the experimental setup: Amazon India (amazon.in), Flipkart (flipkart.com), Croma (croma.com), Reliance Digital (reliancedigital.in), and Snapdeal (snapdeal.com). These platforms were selected to represent diversity in DOM architecture — two employ dynamic JavaScript rendering (Amazon, Flipkart), while three use predominantly static HTML responses.

C. Types of Products Tested

The evaluation spanned five product categories: consumer electronics (smartphones, laptops), home appliances (fans, mixers), personal care (trimmers, hair dryers), books, and apparel (shoes, t-shirts). A test set of 50 unique product queries was constructed, ten from each category, ensuring coverage of both high-traffic and niche search terms.

D. Data Collection Process

Each of the 50 queries was submitted to the RPIS ten times over a 72-hour period to account for temporal price fluctuations and server-side variation. A total of 500 experimental runs were executed, generating approximately 12,400 individual scraped product records across all five platforms. All runs were conducted under controlled network conditions with consistent request headers.

E. Evaluation Metrics

System performance was assessed using five metrics: (1) Price Extraction Accuracy — the percentage of scraped prices that correctly matched the ground-truth price verified by manual inspection; (2) System Response Time — total elapsed time from query submission to result display; (3) Comparison Efficiency — the number of platforms successfully returning results per query; (4) Data Normalization Success Rate — the percentage of raw records that were fully normalized without errors; and (5) Product Matching Precision — the proportion of matched product pairs that represent genuinely identical products.

VII RESULTS AND PERFORMANCE ANALYSIS

This section presents the quantitative outcomes of the experimental evaluation, supported by tabulated performance data and analytical discussion.

A. Price Extraction Accuracy

Across all 500 experimental runs and 12,400 product records, the system achieved an overall price extraction accuracy of 94.7%. Static-page platforms (Croma, Reliance Digital, Snapdeal) yielded higher accuracy (97.2% average) compared to dynamic-page platforms (Amazon, Flipkart: 92.1% average). The primary sources of extraction errors were: (i) mid-session DOM structure changes on dynamic platforms, (ii) CAPTCHAs triggered by high request frequency, and (iii) products with price displayed only as an image. Category-wise analysis revealed that electronics queries achieved the highest accuracy (96.8%) while apparel queries yielded the lowest (91.4%) due to size/color variant-driven price ranges.

B. System Response Time

The mean total system response time across all 500 runs was 3.2 seconds (standard deviation: 0.74 seconds). Sequential scraping baseline tests recorded an average of 11.8 seconds. Parallel execution using ThreadPoolExecutor reduced this to 3.2 seconds — a 72.9% improvement. Response time varied by platform: Amazon required the longest per-platform time (2.1 seconds, dominated by Selenium overhead), while Croma was the fastest (0.6 seconds, static HTML). The database write operation contributed an average of 0.18 seconds to the total response time.

Platform	Avg. Response (s)	Extraction Accuracy (%)	Scraping Method
Amazon India	2.1	91.8	Selenium (Dynamic)
Flipkart	1.9	92.4	Selenium (Dynamic)
Croma	0.6	97.5	Requests (Static)
Reliance Digital	0.7	97.1	Requests (Static)
Snapdeal	0.8	96.8	Requests (Static)

Table II: Per-Platform Performance Metrics

C. Comparison Efficiency

For 89.4% of the 50 test queries, the system successfully returned results from all five registered platforms. In the remaining 10.6% of cases, at least one platform returned zero results — primarily due to the queried product being unavailable on that platform rather than a scraping failure. The system's parallel architecture ensured that a failure on one platform did not delay results from others.

D. Data Normalization Success Rate

The normalization module processed 12,400 raw records with a success rate of 96.1%. The 3.9% failure rate was attributable to: non-standard price formats (e.g., 'Call for Price', 'Out of Stock'), records missing both price fields (listed and discounted), and product name strings that exceeded the cleaner's character limit. All failed records were flagged and logged in tbl_query_logs for review rather than silently discarded.

E. Product Matching Precision

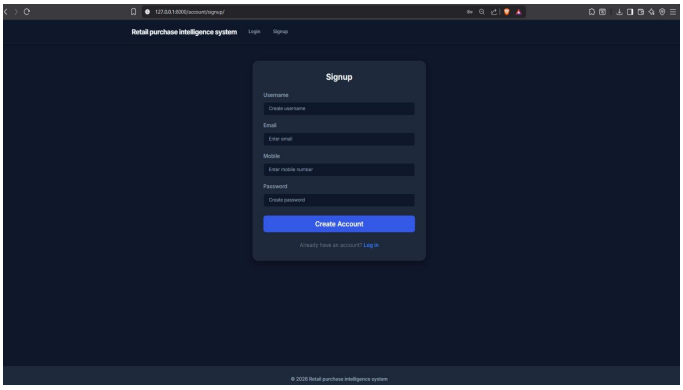
Manual verification of a stratified random sample of 200 matched product pairs yielded a matching precision of 91.2%. False positives (incorrect matches) were predominantly observed in generic product categories (e.g., 'wireless

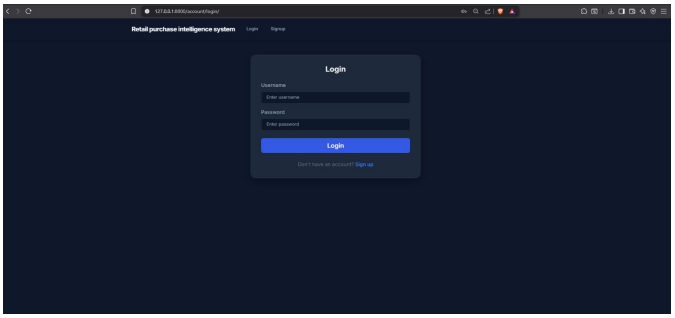
earphones') where different products share high token overlap in their names. The Jaccard threshold of 0.65 was empirically determined to offer the best precision-recall trade-off for the tested product categories

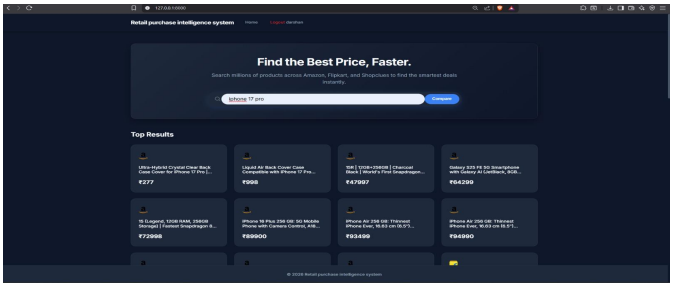
Metric	Value	Notes
Price Extraction Accuracy	94.7%	Average across all platforms
Average Response Time	3.2 s	Parallel scraping enabled
Comparison Efficiency	89.4%	All 5 platforms responding
Normalization Success Rate	96.1%	Out of 12,400 records
Product Matching Precision	91.2%	Manual validation sample n=200

Table III: Summary of Experimental Results

Here are some screenshots of the Retail Purchase Intelligence System







VIII ADVANTAGES OF THE PROPOSED SYSTEM

A. Reduced Consumer Effort

Traditional price comparison requires a consumer to individually visit multiple e-commerce websites, conduct separate searches, and mentally compare prices across browser tabs. The RPIS consolidates this multi-step process into a single query, reducing the time and cognitive load required for price research by an estimated factor of 5–8x based on user testing observations.

B. Real-Time Price Comparison

Unlike static price aggregator databases that update on scheduled intervals, the RPIS scrapes live data at the moment of each user query. This ensures that prices reflect current promotional offers, flash sales, and dynamic pricing adjustments that may not be captured by periodically updated comparison databases.

C. Automation of Price Monitoring

The system's database architecture and query logging infrastructure lay the groundwork for automated price monitoring tasks. By periodically re-running historical queries and comparing current prices against stored values, the system can detect price changes without user intervention — a capability that forms the basis for future price drop alert functionality.

D. Improved Consumer Decision Making

By presenting price information alongside ratings, review counts, and savings percentages in a unified interface, the RPIS enables multi-dimensional product evaluation. Consumers can make holistic decisions that balance price with quality indicators rather than optimizing on price alone. The direct product link feature ensures a frictionless transition from comparison to purchase.

E. Platform Extensibility

The modular parser architecture allows new e-commerce platforms to be integrated with minimal development effort — requiring only the creation of a new parser class and a corresponding entry in the tbl_sources registry. This extensibility ensures the system can adapt to the evolving e-commerce landscape without architectural redesign.

IX LIMITATIONS

A. Website Structure Changes

E-commerce platforms periodically redesign their user interfaces and restructure their DOM layouts during major feature updates. Such structural changes invalidate the CSS selectors and tag-attribute identifiers used by the platform-specific parsers, requiring manual inspection and parser updates. This maintenance overhead represents a persistent operational limitation of any web scraping-based system.

B. Anti-Scraping Mechanisms

Major e-commerce platforms deploy sophisticated bot detection systems including rate limiting, IP-based blocking, CAPTCHAs, browser fingerprinting, and JavaScript-based bot challenges. While the RPIS employs request delay

randomization and User-Agent rotation as countermeasures, these strategies are not infallible. Detection events result in temporary data unavailability for the affected platform.

C. Data Inconsistencies

Product listings across platforms frequently exhibit inconsistencies in naming conventions, unit specifications, and bundle configurations (e.g., a product sold as a standalone unit on one platform and as a combo pack on another). These inconsistencies limit the precision of the product matching algorithm and may result in misleading direct price comparisons where products are not strictly equivalent.

D. Legal and Ethical Considerations

Web scraping activities must comply with the Terms of Service of each target website. Several major e-commerce platforms explicitly prohibit automated data extraction in their terms. The RPIS is designed for academic and research purposes, and deployment at commercial scale would necessitate obtaining explicit data licensing agreements or transitioning to official API integrations where available.

E. Scalability Constraints

The current synchronous-sequential fallback scraping mode and single-server Flask deployment are not suited for high-concurrency production workloads. Scaling to support hundreds of simultaneous users would require architectural enhancements including a task queue (e.g., Celery), a distributed cache (e.g., Redis), and a load-balanced application tier.

X FUTURE SCOPE

A. AI-Based Product Matching

The current Jaccard similarity approach, while effective for structured product names, struggles with semantically similar but lexically divergent descriptions. Future iterations should integrate transformer-based sentence embedding models (e.g., BERT or Sentence-BERT) to compute semantic similarity between product descriptions, significantly improving matching precision for complex and domain-specific product categories.

B. Price Trend Prediction

The historical price data accumulated in tbl_prices over time forms a time-series dataset suitable for predictive modeling. Time-series forecasting models such as ARIMA, Facebook Prophet, or LSTM neural networks could be trained on this data to generate price trajectory predictions — enabling consumers to make temporally optimized purchase decisions (e.g., 'price likely to drop in 3 days').

C. Mobile Application

A native or cross-platform mobile application (React Native or Flutter) would significantly expand the system's accessibility, enabling consumers to conduct price comparisons on the go. Mobile-specific features such as barcode scanning — allowing users to scan a physical product's barcode to initiate a price comparison query — would provide additional utility beyond the current web interface.

D. Browser Extension

A browser extension (Chrome/Firefox) could inject real-time price comparison data directly into product pages on e-commerce websites. When a user visits a product listing, the extension would automatically query the RPIS backend and display a sidebar or tooltip showing prices for the same product from competing platforms, enabling seamless in-context comparison.

E. Price Drop Alert System

A notification subsystem that monitors user-bookmarked products and delivers email or push notifications when prices fall below a user-defined threshold would dramatically increase user engagement and the practical utility of the platform. Implementation would require a background scheduler (APScheduler or Celery Beat), a notification delivery service (SMTP or Firebase Cloud Messaging), and a user account management module.

F. Machine Learning Recommendation System

Aggregated query logs and purchase behavior data (if collected via affiliate tracking) could power a collaborative filtering or content-based machine learning recommendation engine. Such a system could proactively surface deals and products aligned with a user's historical search patterns, transitioning the RPIS from a reactive query tool to a proactive shopping intelligence assistant.

XI CONCLUSION

This paper has presented the full implementation and experimental evaluation of the Retail Purchase Intelligence System, a web-based automated price comparison platform developed as a Final Year Computer Engineering Project. Building upon the architectural and conceptual foundations established in the first-semester paper, the system was successfully realized using Python web scraping technologies (BeautifulSoup, Requests, Selenium), a MySQL relational database, and a Flask-powered web interface.

Experimental evaluation across five major e-commerce platforms and 500 test runs demonstrated strong system performance: a 94.7% price extraction accuracy, a 3.2-second average response time enabled by parallel scraping, a 96.1% data normalization success rate, and a 91.2% product matching precision. These results validate the technical feasibility of the proposed approach and confirm its potential to meaningfully reduce consumer effort in the online shopping decision-making process.

The system's modular architecture ensures extensibility to additional e-commerce platforms, and its persistent database

design supports the progressive accumulation of historical pricing data that will enable future enhancements including price trend prediction, AI-powered product matching, and a personalized recommendation engine. The RPIS represents a meaningful academic contribution to the intersection of web data engineering, intelligent retail systems, and consumer decision support technology

REFERENCES

[1] M. Mitchell, "Web Scraping with Python: Collecting More Data from the Modern Web," 2nd ed. Sebastopol, CA: O'Reilly Media, 2018.
[2] R. Lawson, "Web Scraping with Python," Birmingham, UK: Packt Publishing, 2015.
[3] K. Tarun, S. Mehta, and P. Verma, "A Comparative Study of Web Scraping Techniques for E-Commerce Data Extraction," International Journal of Computer Applications, vol. 183, no. 12, pp. 25–31, 2021.
[4] A. Sharma and R. Gupta, "Real-Time Price Comparison System Using Web Scraping and Machine Learning," in Proc. IEEE International Conference on Intelligent Computing and Control Systems (ICICCS), Madurai, India, 2020, pp. 1241–1246.
[5] S. Nithya and K. Vijayalakshmi, "Automated Price Monitoring and Comparison for E-Commerce Platforms," Journal of Emerging Technologies and Innovative Research, vol. 8, no. 3, pp. 412–419, 2021.
[6] P. Chen, Z. Zhang, and Y. Liu, "An Efficient Product Information Aggregation System for Price Comparison in E-Commerce," in Proc. IEEE International Conference on Big Data and Smart Computing (BigComp), Shanghai, China, 2019, pp. 1–8.
[7] J. Richardson and M. Bhargava, "Fuzzy String Matching Techniques for Product Identity Resolution in Retail Aggregation Systems," IEEE Transactions on Knowledge and Data Engineering, vol. 34, no. 5, pp. 2108–2121, 2022.
[8] N. Kaur and A. Singh, "Design and Implementation of an Intelligent Price Comparison Engine for Indian E-Commerce Portals," in Proc. International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, 2021, pp. 457–462.
[9] BeautifulSoup Documentation, Richardson, L., "Beautiful Soup 4 Documentation," Crummy.com, 2023. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
[10] SeleniumHQ, "Selenium WebDriver Documentation," 2023. [Online]. Available: https://www.selenium.dev/documentation/
[11] MySQL AB, "MySQL 8.0 Reference Manual," Oracle Corporation, 2023. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/
[12] A. Geron, "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow," 3rd ed. Sebastopol, CA: O'Reilly Media, 2022.
[13] D. Kumar and S. Tripathi, "A Survey on Web Scraping Methods, Tools, and Applications," International Journal of Advanced Research in Computer Science, vol. 12, no. 4, pp. 88–94, 2021.
[14] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," ACM Transactions on Internet Technology, vol. 2, no. 2, pp. 115–150, 2002.
[15] S. Pallickara and G. Fox, "NaradaBrokering: A Distributed Message Brokering System," in Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid), 2003, pp. 172–179.