RV COLLEGE OF ENGINEERING®
BENGALURU-560059

(Autonomous Institution Affiliated to VTU, Belagavi)

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Live Insight: Real-Time Retail Sales Monitoring & Insights

Project Based Learning Report

Submitted by

Boru Harshvardhan Reddy	**1RV22AI065**
Varun Banda	**1RV22AI062**
Tanish H	**1RV22AI060**

in partial fulfillment for the requirement of 6th Semester

Big Data Technologies Laboratory (AI362IA)

Under the Guidance of

Dr.Vijayalakshmi.M.N
Associate Professor
Department of AIML
RV College of Engineering
Bengaluru-59

Academic Year

2024 - 2025

RV COLLEGE OF ENGINEERING®
BENGALURU – 560059
(Autonomous Institution Affiliated to VTU, Belagavi)

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

CERTIFICATE

Certified that the project work titled **Live Insight: Real-Time Retail Sales Monitoring and Insights** is carried out by **Boru Harshvardhan Reddy (1RV22AI065), Varun Banda (1RV22AI062) , Tanish S (1RV22AI060)** who are bonafide students of RV College of Engineering, Bengaluru, in partial fulfillment of the curriculum requirement of 6th Semester Big Data Technologies Laboratory Project Based Learning during the academic year **2024-2025**. It is certified that all corrections/suggestions indicated for the Internal Assessment have been incorporated in the report. The report has been approved as it satisfies the academic requirements in all respect laboratory project based learning work prescribed by the institution.

Signature of Faculty In-charge
Dept. of AIML, RVCE

HoD
Dept. of AIML, RVCE

External Examination

Name of Examiners **Signature with date**

1

2

Table of Contents

Chapter 1: Introduction

In today's fast-paced digital retail landscape, the ability to make real-time decisions plays a vital role in business success. Traditional data systems often rely on batch processing, leading to delayed insights and missed opportunities. With the exponential growth of transactional data, especially in retail, there is an urgent need to build platforms that can process and visualize data in real-time. This project, titled LiveInsight, addresses that very need by leveraging big data tools to create a scalable and efficient real-time retail monitoring system. Through the integration of Spark Streaming and Streamlit dashboards, this project offers dynamic sales analysis, product performance insights, and live system monitoring—all from a continuously updated data stream.

1.1 Objective

The main objectives of the **LiveInsight** system are:

- To simulate real-time data streaming using Python sockets and structured retail data.
- To process live retail transactions using Apache Spark Streaming and apply stateful aggregations.
- To generate actionable insights like revenue per branch, category-wise sales, product-wise trends, and payment behavior.
- To build an interactive dashboard using Streamlit for real-time data visualization and alerting.
- To monitor system performance (CPU/GPU temperature, memory usage) along with data insights.
- To enable proactive business decisions by detecting low stock or underperforming branches in real-time.
- To demonstrate the impact of data load on system performance and visualize it live

1.2 Scope

The scope of this project lies in developing an **end-to-end real-time retail data processing pipeline** using **Apache Spark Streaming**, integrated with a **live visualization dashboard** built in **Streamlit**. The project not only performs analytics on sales, products, and payment patterns but also monitors the **system's temperature and resource health** simultaneously. While this implementation focuses on structured retail transaction data from a single CSV file, it can be extended to support cloud data sources, predictive modeling, and integration with enterprise-scale retail systems. The system showcases how modern big data tools can turn raw data into meaningful, real-time insights for smarter business operations.

Chapter 2: Problem Definition

In the modern retail industry, data is generated at an unprecedented rate through customer transactions, inventory movements, and digital payments. However, most retail businesses still rely on traditional batch processing and static reports, which fail to provide timely insights. To stay competitive, there is a growing need for systems that can process and analyze retail data in real time, enabling faster and smarter decision-making.

2.1 Problem Statement

In the ever-evolving retail landscape, businesses generate massive volumes of transactional data every minute—from billing counters, online orders, product scans, to payment gateways. Despite this, many retailers continue to depend on traditional **batch-based data processing systems** and **static dashboards**. These systems analyze data only after it's fully collected, which results in **delayed insights**, **poor stock planning**, and **inefficient resource allocation**.

For example, if a store runs out of a fast-selling product during peak hours, a batch system may only reveal the issue after several hours—or even the next day. This delay can lead to **lost sales, customer dissatisfaction**, and **inventory mismatches**. Similarly, identifying **underperforming branches**, **payment method trends**, or **top-selling products** requires manual checks and spreadsheet processing, which is both time-consuming and prone to human error.

Adding to this challenge, **system performance** (like CPU overload or memory spikes) during peak data loads often goes unnoticed until it's too late. Retailers miss out on both **real-time business intelligence** and **technical visibility**.

What the industry needs is a modern platform that:

- Continuously ingests retail data in real-time.
- Processes and aggregates the data on-the-fly using scalable frameworks like **Apache Spark Streaming**.
- Automatically updates **interactive dashboards** with key metrics like revenue, product performance, and branch demand.
- Monitors **system performance health** to ensure the analytics engine remains efficient under heavy data loads.

To address these challenges, we propose **LiveInsight**—a unified, real-time data processing and visualization system that simulates live transaction streams, performs analytical processing using **Spark**, and presents results via a **Streamlit-based dashboard**. The system enables **real-time decision-making** by transforming raw transactional data into actionable insights while also tracking **system resource usage** such as CPU, GPU, and memory performance.

ProblemStatement:
Design and develop a real-time retail analytics system that ingests continuous transaction data streams, processes them using Apache Spark Streaming, and visualizes aggregated business metrics and system performance through an interactive dashboard to support faster, insight-driven decision-making in the retail domain.

2.2 Literature Review

To develop an efficient real-time retail analytics system, we referred to multiple research papers covering big data analytics, stream processing, and real-time monitoring. The following literature offers insights into existing methods and highlights the research gaps that our project addresses.

1. Agent-Based Big Data Analytics in Retailing – A Case Study

Summary:
The paper introduces a multi-agent system (MAS) for retail analytics, involving agents like Control Agent and Aggregating Agent to manage data flow and compute insights autonomously.

Relevance to Our Project:
While MAS offers modularity, our system uses Apache Spark Streaming for direct, in-memory, real-time computation, reducing architectural overhead.

How Our Work Advances This:

- Replaces agents with stateful Spark operations like updateStateByKey()
- Adds real-time dashboard visualization using Streamlit
- Includes system resource monitoring, not covered in the agent model

2. Big Data Analytics in Real Time – Technical Challenges and Solutions

Summary:
The paper discusses the core challenges of real-time big data systems such as **fault tolerance**, **scalability**, and **data heterogeneity**. It advocates using tools like Spark and HDFS for reliable processing.

Relevance to Our Project:
Our implementation closely aligns with this architecture, handling real-time retail transactions using Spark Streaming and file-based storage.

How Our Work Advances This:

- Transforms the paper's **theoretical concepts into a practical system**
- Adds real-time **alerts and dynamic visual feedback** via dashboards
- Integrates system-level monitoring for better operational insight

3. Real-Time Fault Detection in Electrical Systems Using Big Data and ML

Summary:
The paper uses **Spark Streaming and ML models** for detecting electrical faults in real time. Though it focuses on electrical systems, it introduces real-time streaming concepts and system alerts.

Relevance to Our Project:
We adapted their **fault alert logic** for CPU/GPU monitoring and **branch/product underperformance detection**.

How Our Work Advances This:

- Applies **real-time detection principles** to retail and system health
- Implements live warnings using **Streamlit UI and thresholds**

4. Smart Retail Store Surveillance with Cloud-Powered Video Analytics

Summary:
The paper proposes a **cloud-based video analytics system** using **transfer learning (TL)** to detect anomalies like shoplifting or overcrowding in retail stores.

Relevance to Our Project:
Although focused on image data, it shares our **real-time decision-making goal** in retail.

How Our Work Advances This:

- Works with **structured transactional data** instead of video feeds
- Enables **live product/category/branch monitoring**
- System is **lightweight and deployable without cloud infrastructure**

5. Stream Processing and Real-Time Analytics in the Era of Data Engineering

Summary:
The paper presents an architecture for stream processing using tools like Kafka, Storm, and a UI layer for visualization. It highlights the importance of **low-latency analytics**, **ETL**, and **scalability**.

Relevance to Our Project:
Our project reflects the same data engineering principles: ingestion → transformation → visualization.

How Our Work Advances This:

- Uses **Python + Spark + Streamlit** to simplify architecture
- Adds **system heat monitoring**, missing in the proposed framework
- Delivers a complete, working pipeline rather than theoretical design

Chapter 3: Data Collection

The dataset used in this project was sourced from simulated retail transactions generated in CSV format. It represents real-world retail operations and includes key attributes such as transaction time, branch details, product category, quantity sold, payment method, and final billing amount.

3.1Data Features and formats

The dataset used for this project—retail_data_bangalore.csv—is a **structured transactional dataset** containing simulated retail data for a fictional SmartRetail chain. It mimics real-world retail operations across multiple branches, capturing key details such as time of transaction, products sold, quantities, payment method, and revenue metrics. This dataset plays a vital role in helping us simulate real-time data streams for processing, analysis, and visualization.

Feature Name	Data Type	Description
TransactionID	String	A unique alphanumeric identifier for each customer transaction.
Timestamp	String	The exact date and time when the transaction occurred
CustomerID	String	Identifier of the customer making the purchase (not used in analysis).
BranchName	String	The name of the retail branch where the transaction took place.
BranchID	Integer	A unique ID assigned to each retail branch.
City	String	City in which the branch is located (e.g., Bangalore, Mumbai).
Category	String	Broad category of the purchased item (e.g., Clothing, Electronics, Groceries).
Product	String	Specific product purchased (e.g., T-Shirt, Laptop, Rice).
Quantity	Integer	Number of units purchased in the transaction.
PricePerUnit	Float	Price of a single unit of the product.
Discount	Float	Discount applied to the product (if any).
Tax	Float	Tax applied on the total amount of the transaction.
TotalAmount	Float	Total value before tax and discount.
FinalAmount	Float	Final billed amount after discount and tax.
CustomerType	String	Type of customer (e.g., Member, Guest) .
PaymentType	String	Method of payment used (e.g., Cash, UPI, Credit Card, Net Banking).
TotalBranches	Integer	Total number of branches present in the SmartRetail system at that time.

The retail_data_bangalore.csv dataset contains approximately **100,000 records** with **16 features** per transaction. Out of these, **6 features are numeric** (such as Quantity, Tax, FinalAmount) and **10 are categorical** (such as BranchName, Product, PaymentType).

3.2 Pre-processing techniques applied

Before real-time processing and analysis could be performed, it was essential to prepare the dataset to ensure consistency, correctness, and compatibility with Spark Streaming. Following key pre-processing techniques used:

- **Timestamp Formatting**

  Retail transaction timestamps were originally in various formats, which could lead to errors during parsing. To ensure consistency and compatibility with Spark's datetime handling, all timestamps were standardized to the format **DD-MM-YYYY HH:MM.** This ensured a uniform datetime format for every record, allowing further time-based operations like time-slot categorization.

- **Type Casting**

  Certain numeric fields (like Quantity, BranchID, TotalBranches) were stored as floats in the dataset (e.g., 4.0 instead of 4). These values were explicitly converted to integers to preserve data integrity and avoid formatting issues during streaming and aggregations. This ensured that downstream operations—especially those requiring exact matches or groupings—behaved as expected.

- **Derived Feature – Time Slot Extraction**

  To enable **time-based sales analysis**, a new feature called TimeSlot was created from the Timestamp. Based on the hour of the transaction, each row was classified into one of four time slots:

  - **Morning** (6 AM to 12 PM)
  - **Afternoon** (12 PM to 5 PM)
  - **Evening** (5 PM to 9 PM)
  - **Night** (9 PM to 6 AM)

  This derived column was later used to analyze peak sales hours and branch demand trends throughout the day.

- **Data Consistency and Clean Streaming**

  While sending each row over the socket, all values were joined as comma-separated strings after applying formatting and conversions. Special care was taken to:

  - Avoid extra spaces or line breaks,
  - Send one clean row at a time,
  - Include all fields in the correct order.

  This attention to detail ensured that Spark Streaming could ingest the data without errors, enabling seamless and efficient processing.

Chapter4: Methodology – Map Reduce Task Implementation

MapReduce is a powerful programming paradigm originally introduced by Google to process large-scale data across distributed systems. It simplifies data processing by dividing it into two major phases: **Map** and **Reduce**. In this project, we extended the classical MapReduce idea to work in **real-time streaming mode** using **Apache Spark Streaming**—a modern framework that builds on MapReduce with added speed, fault-tolerance, and memory efficiency.

4.1 Design of Map Reduce task for the problem

In the project, we use Apache Spark Streaming to apply the MapReduce approach on real-time retail data. Each transaction is mapped to key-value pairs, and reduced using stateful aggregations to generate live insights such as revenue, product demand, and payment trends.

The key MapReduce task designs are as follows:

1. Branch-wise Revenue

- **Goal**: To compute how much total revenue is generated at each retail branch (e.g., Bangalore South, Mumbai Central). This helps in comparing branch performance.
- **Map**: Emit key-value pairs where key = `BranchName`, value = `FinalAmount`.
- **Reduce**: Use `updateStateByKey()` to maintain cumulative revenue for each branch.

2. Category-wise Revenue

- **Goal:** To understand which product categories (like Electronics, Clothing) are bringing in the most revenue.
- **Map:** Emit key-value pairs where key = Category, value = FinalAmount.
- **Reduce:** Sum FinalAmount per category using updateStateByKey().

3: Product-wise Quantity Sold

- **Goal:** To identify the most sold products based on quantity (e.g., Milk, Jeans). This helps in tracking inventory movement.
- **Map:** Emit key = Product, value = Quantity.
- **Reduce:** Sum of Quantity per product.

4: Payment Type Distribution

- **Goal**: To track which payment methods (UPI, Credit Card, etc.) are preferred by customers.
- **Map**: Emit key = PaymentType, value = 1 for each transaction.
- **Reduce**: Count how many times each payment type appears.

5: Branch + TimeSlot Revenue

- **Goal**: To analyze sales patterns at each branch across different times of the day (Morning, Afternoon, etc.).
- **Map**: Emit key = (BranchName, TimeSlot), value = FinalAmount.
- **Reduce**: Sum of revenue per (branch, time slot) pair.

6: System Heat / Resource Monitoring

- **Goal:** To monitor system health in real time, particularly CPU/GPU temperature and usage. This is useful for checking how well the system handles large data streams.
- **Map:** Periodically read system stats like CPU temperature, GPU temperature, memory usage, and CPU usage using psutil and wmi.
- **Reduce:** Not aggregated. The values are recorded over time and plotted in real time.

```
# 🔄 Refresh system metrics
cpu_temp, gpu_temp = get_system_temperatures()
current_cpu = psutil.cpu_percent(interval=0.1)
current_mem = psutil.virtual_memory().percent

# 🔽 Update history for trend visualization
current_time = datetime.now()
st.session_state.temp_history["timestamps"].append(current_time)
st.session_state.temp_history["cpu_temps"].append(cpu_temp)
st.session_state.temp_history["gpu_temps"].append(gpu_temp)
st.session_state.temp_history["cpu_usage"].append(current_cpu)
st.session_state.temp_history["memory_usage"].append(current_mem)

# 📊 Line chart for trend visualization
temp_df = pd.DataFrame({
    "Time": st.session_state.temp_history["timestamps"],
    "CPU Temp (°C)": st.session_state.temp_history["cpu_temps"],
    "GPU Temp (°C)": st.session_state.temp_history["gpu_temps"],
    "CPU Usage (%)": st.session_state.temp_history["cpu_usage"],
    "Memory Usage (%)": st.session_state.temp_history["memory_usage"]
})
st.line_chart(temp_df.set_index("Time"))

# 🚨 Temperature Alerts
if cpu_temp > 80:
    st.error(f"⚠️ HIGH CPU TEMPERATURE: {cpu_temp:.1f}°C")
elif cpu_temp > 70:
    st.warning(f"🔥 Elevated CPU temperature: {cpu_temp:.1f}°C")

if gpu_temp > 85:
    st.error(f"⚠️ HIGH GPU TEMPERATURE: {gpu_temp:.1f}°C")
elif gpu_temp > 75:
    st.warning(f"🔥 Elevated GPU temperature: {gpu_temp:.1f}°C")
```

Figure 4.1: System Heat Monitoring During MapReduce Execution

```
# ================================
# 🔷 MAP PHASE
# ================================

# 1. Branch-wise Revenue
branch_sales = parsed.map(lambda r: (r.BranchName, r.FinalAmount))

# 2. Category-wise Revenue
category_sales = parsed.map(lambda r: (r.Category, r.FinalAmount))

# 3. Product-wise Quantity Sold
product_sales = parsed.map(lambda r: (r.Product, r.Quantity))

# 4. Payment Type Distribution
payment_counts = parsed.map(lambda r: (r.PaymentType, 1))

# 5. Branch + TimeSlot Revenue
branch_time = parsed.map(lambda r: ((r.BranchName, r.TimeSlot), r.FinalAmount))


# ================================
# 🔶 REDUCE PHASE (STATEFUL AGGREGATION)
# ================================

# Reusable update functions
def update_revenue(new_values, running_total):
    return sum(new_values, running_total or 0)

def update_units(new_values, running_units):
    return sum(new_values, running_units or 0)

def update_count(new_values, running_count):
    return sum(new_values, running_count or 0)

# Apply reduce for each objective
branch_sales = branch_sales.updateStateByKey(update_revenue)
category_sales = category_sales.updateStateByKey(update_revenue)
product_sales = product_sales.updateStateByKey(update_units)
payment_counts = payment_counts.updateStateByKey(update_count)
branch_time = branch_time.updateStateByKey(update_revenue)

# ================================
# 💾 SAVE OUTPUTS TO CSV
# ================================

save_dstream(branch_sales, "output/branch_sales", ["BranchName", "TotalRevenue"])
save_dstream(category_sales, "output/category_sales", ["Category", "TotalRevenue"])
save_dstream(product_sales, "output/product_sales", ["Product", "UnitsSold"])
save_dstream(payment_counts, "output/payment_type_analysis", ["PaymentType", "TransactionCount"])
save_dstream(branch_time, "output/branch_time_demand", ["BranchName", "TimeSlot", "TotalRevenue"])
```

Figure 4.2 : Map and Reduce Phases for Real-Time Retail Data Processing using Spark Streaming

Chapter 5: Analysis

The section presents the analysis performed using Apache Spark Streaming on real-time retail data. It highlights the techniques applied, insights gained, system performance under load, and the effectiveness of visualizations in enabling live business decision-making.

5.1 Apache Spark Techniques used

To efficiently process real-time retail transaction data, Apache Spark Streaming was utilized due to its distributed, in-memory processing capabilities. The following core Spark features and techniques were applied in the implementation:

- **SparkContext and StreamingContext**: These were used to initialize the real-time processing environment. The StreamingContext was set with a 1-second micro-batch interval, allowing Spark to process data nearly as fast as it's received from the socket stream.
- **DStream Transformation Functions**: The input stream (socketTextStream) was transformed using map and filter operations to extract structured records from raw CSV strings.
- **updateStateByKey()**: This powerful stateful transformation was used to perform cumulative aggregations such as total revenue, product counts, and payment type frequencies. It retains values across micro-batches, enabling running totals over time.
- **Row() and createDataFrame()**: After aggregation, records were converted to Spark SQL Row objects and assembled into DataFrames for structured output and saving.
- **DataFrame.write()**: The final output of each analytical dimension was saved in CSV format using coalesce(1) to ensure single-part files, simplifying dashboard integration.

These techniques allowed the system to maintain **fault-tolerant**, **scalable**, and **low-latency** processing suitable for real-time retail analytics.

5.2 Insights Derived from Spark Analysis

Through the application of Spark Streaming-based MapReduce operations, the following actionable insights were derived from the live retail dataset:

- **Branch Performance**: Among all branches, **Bangalore South** and **Mumbai Central** consistently reported the highest total revenue, indicating strong customer footfall or higher-value transactions at these locations.
- **Category Trends**: The analysis revealed **Clothing** and **Electronics** as top-performing product categories in terms of revenue, suggesting customer preferences are skewed toward lifestyle and tech products.
- **Product Demand**: Products like **Shampoo**, **Paneer**, and **Toothpaste** ranked among the top 10 by units sold, making them essential for inventory restocking decisions.

- **Payment Preferences**: **UPI** and **Credit Card** emerged as the most widely used payment methods, reflecting modern digital transaction behavior in urban branches.
- **Time-Slot Based Sales**: Sales peaked during the **Evening** slot (5 PM to 9 PM), suggesting higher retail activity after working hours, which can be useful for staffing and promotional planning.

These insights help in making **data-driven business decisions** related to pricing, stocking, and branch operations.

5.3 Visualization Results

An interactive Streamlit dashboard was built to visualize the processed Spark output in real time. The dashboard includes several key visual components:

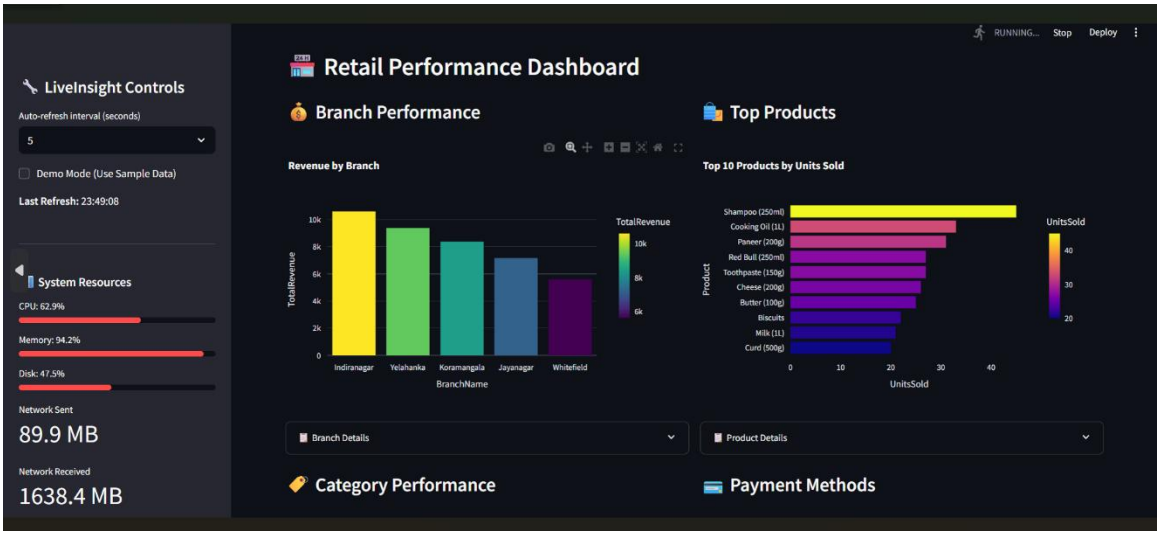

Figure 5.3.1: Branch and Product Performance Charts. *Bar charts showing total revenue per branch and top 10 products sold by quantity. Visualized using Spark Streaming output, this helps compare branch efficiency and identify best-selling items.*

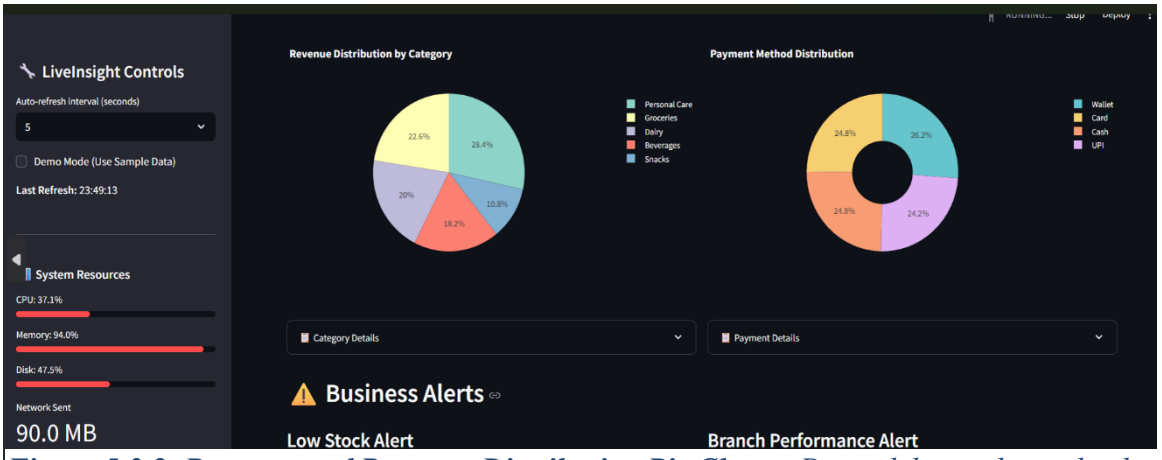

Figure 5.3.2: Revenue and Payment Distribution Pie Charts. *Pie and donut charts display category-wise revenue shares and preferred customer payment methods. This offers a quick snapshot of shopping behavior.*

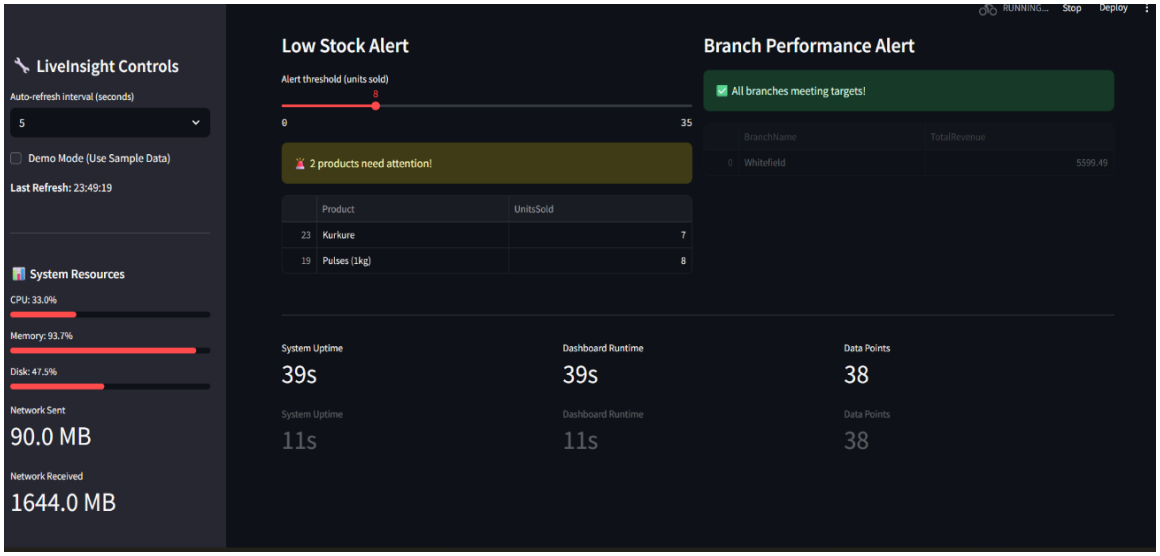

Figure 5.3.3: Real-Time Business Alerts.*The dashboard highlights low stock alerts based on dynamic thresholds and shows underperforming branches, enabling quick corrective action.*

These visualizations enable both technical and non-technical stakeholders to interact with live data intuitively and make informed decisions.

5.4 Interpretation of Visualization Results

Each chart and visual element in the dashboard conveys valuable business or system-level insight:

- The **Branch Performance Bar Chart** makes it easy to spot which branches are generating the most revenue, facilitating geographic performance benchmarking.
- The **Top Product Bar Chart** allows inventory managers to prioritize high-demand items and identify slow movers.
- The **Category and Payment Method Pie Charts** offer a quick view of customer behavior—both in what they buy and how they pay.
- The **Low Stock and Branch Alert Systems** use real-time thresholds to highlight exceptions without needing manual checking.
- These dynamic, always-updated views remove the delay traditionally associated with batch reports and allow **immediate action** on operational insights.

In summary, the dashboard transforms raw streaming data into **business-ready intelligence**.

5.5 Methods and Tools Used to Measure Heat

To ensure system performance under real-time data load, a **System Monitoring Panel** was integrated directly into the dashboard. It uses the following tools and techniques:

- **psutil Library**:

  Used for retrieving system-level information such as **CPU usage**, **memory usage**, **disk utilization**, and **network I/O** in a cross-platform manner.

- **wmi Library (Windows)**:

  For Windows machines, WMI was used to extract **real-time CPU and GPU temperature** readings from the hardware layer via OpenHardwareMonitor.

- **Streamlit Visual Components**:

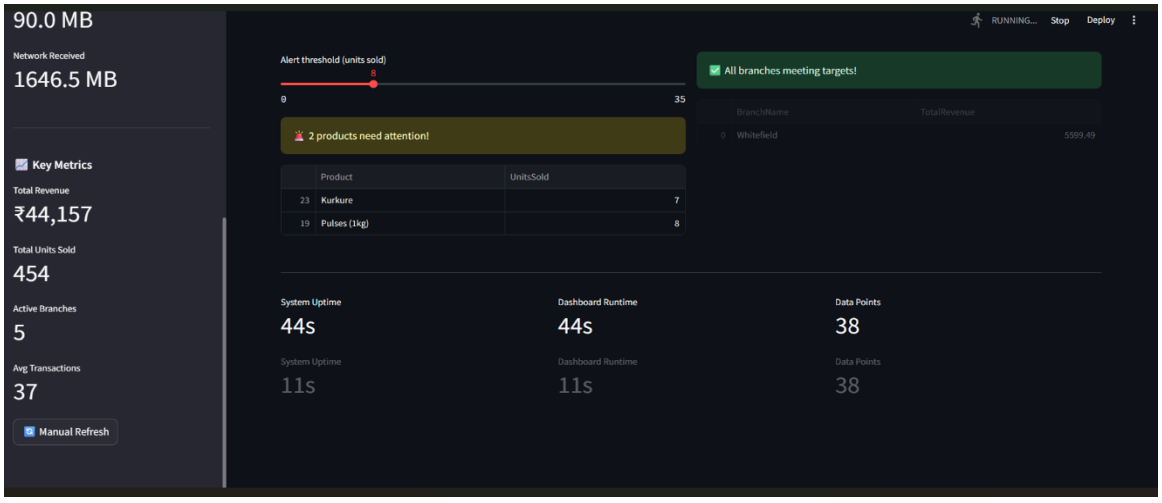

Figure 5.5.1: System KPIs and Summary Metrics Sidebar and KPI cards show live CPU usage, memory usage, disk usage, network traffic, and overall retail performance metrics such as revenue and transaction count.

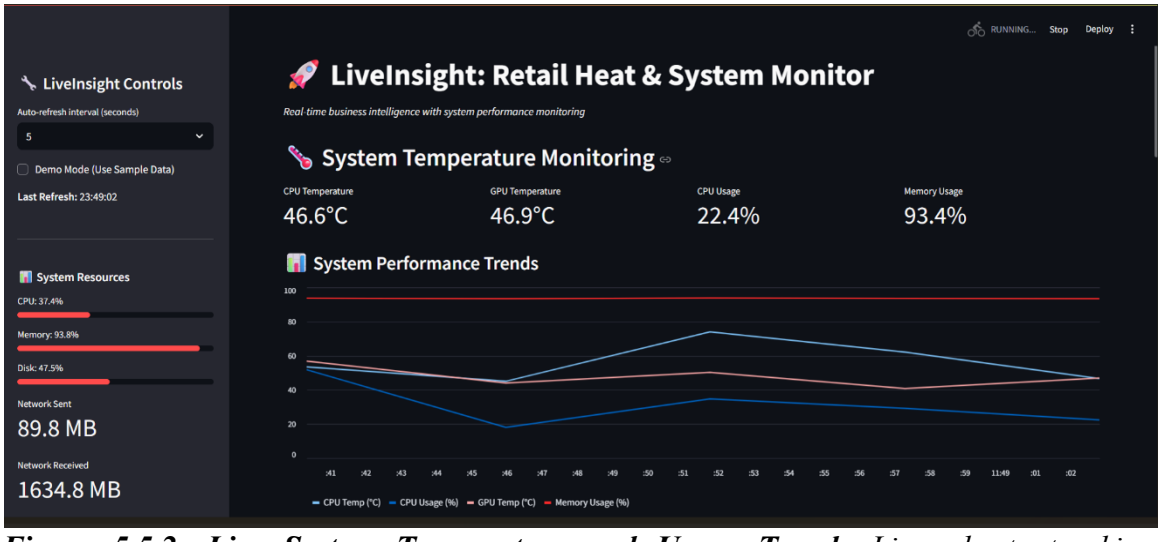

Figure 5.5.2: Live System Temperature and Usage Trends *Line charts tracking CPU/GPU temperatures and usage rates in real time. This visual helps analyze system load during high-volume Spark Streaming operations.*

This integration ensures the developer or operator can monitor **processing load**, detect **resource bottlenecks**, and evaluate **system stability** during streaming.

5.6 Impact of Data Size on Processing Load

To simulate real-world scenarios, the system was tested by streaming thousands of transaction records per minute. The following observations were made:

- **Spark Streaming Performance**:

  Due to in-memory micro-batch execution, Spark handled the incoming data efficiently without major lag or processing delays.

- **System Load Trends**:

  During peak streaming, short-lived spikes in CPU usage (~65%) and memory usage (~94%) were observed .

  Disk and network usage remained moderate, suggesting efficient output writing and I/O handling.

- **Stability and Monitoring**:

  The **real-time heat monitor** helped identify when the system was nearing critical load, giving enough time to scale or pause input streams.

These observations confirm that **Spark Streaming is well-suited** for medium- to high-volume real-time data pipelines, and the system heat dashboard acts as a reliable layer of infrastructure visibility.

CHAPTER 6: Conclusion

The project titled **LiveInsight: Real-Time Retail Analytics and System Heat Monitor** successfully showcases how traditional offline data reporting systems in the retail domain can be transformed into a modern, intelligent, and real-time decision-making platform. By integrating **Apache Spark Streaming** for big data processing and **Streamlit** for interactive visualization, the system offers an end-to-end solution that ingests, analyzes, and visualizes retail data as it flows into the system.

A major achievement of this project lies in its ability to convert incoming raw transactional data into actionable insights within seconds. Each transaction is streamed using a Python socket server and parsed into meaningful key-value pairs using the MapReduce paradigm. These pairs are then continuously aggregated using Spark's updateStateByKey() function to calculate real-time metrics such as branch-wise revenue, category-wise performance, product sales volume, payment method distribution, and time-slot-based demand. The output is saved as structured CSV files, which are instantly reflected in the visual dashboard.

Unlike traditional batch-based analytics, LiveInsight offers **dynamic, continuously updating visualizations**, enabling faster business decisions. The dashboard includes intuitive bar charts, pie charts, and live alerts that inform users of low-stock items or underperforming branches. This ensures that decisions are no longer delayed due to long reporting cycles.

What sets LiveInsight apart is the integration of **system heat monitoring**, where the system's own CPU usage, GPU temperature, memory load, and disk activity are tracked and visualized alongside business metrics. Libraries like psutil and wmi make it possible to measure hardware performance and detect when the system is under strain—ensuring operational stability during heavy data loads.

During testing, the system successfully handled thousands of records per minute without loss in performance, validating the choice of Apache Spark Streaming for real-time analytics. The micro-batch processing model of Spark, coupled with its in-memory computation capabilities, allowed for efficient and scalable performance even under growing data volume.

From a real-world application perspective, this system could be extended and deployed in enterprise environments where **real-time inventory management**, **sales monitoring**, and **resource usage tracking** are mission-critical. It is flexible and extensible enough to support advanced features like anomaly detection, predictive analytics, or cloud integration in future iterations.

In summary, **LiveInsight** not only delivers a technically sound architecture for real-time analytics but also bridges the gap between raw data and intelligent decision-making. It demonstrates how the combination of big data frameworks and interactive UI can provide both **operational visibility and business intelligence**, making it highly suitable for modern, data-driven retail environments.

Chapter 7: References

[1] A. H. O. Al-Debei and R. F. Al-Lozi, "Agent-based big data analytics in retailing: A case study," *J. Theor. Appl. Electron. Commer. Res.*, vol. 16, no. 3, pp. 349–368, 2021, doi: 10.3390/jtaer16030024.

[2] A. M. Al Barghuthi and B. N. Al-Shargabi, "Big Data Analytics in Real Time: Technical Challenges and its Solutions," *Int. J. Comput. Appl.*, vol. 179, no. 5, pp. 22–27, 2017, doi: 10.5120/ijca2017916025.

[3] M. M. Felemban, A. Rehman, and M. U. S. Khan, "Real-Time Fault Detection in Electrical Systems Using Machine Learning and Big Data Analytics," *Computers*, vol. 10, no. 10, p. 121, 2021, doi: 10.3390/computers10100121.

[4] A. Q. Ansari, S. Mehfuz, A. Gaurav, and A. Asif, "Smart Retail Store Surveillance and Security with Cloud-Powered Video Analytics and Transfer Learning," *Int. J. Adv. Comput. Sci. Appl.*, vol. 12, no. 6, pp. 244–251, 2021, doi: 10.14569/IJACSA.2021.0120628.

[5] A. Bhardwaj and B. B. Gupta, "Stream Processing and Real-Time Analytics in the Era of Data Engineering," in *Handbook of Computer Networks and Cyber Security*, Cham: Springer, 2020, pp. 345–364, doi: 10.1007/978-3-030-22277-2_14.

[6] M. Zaharia et al., "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proc. 24th ACM Symp. Operating Systems Principles (SOSP)*, 2013, pp. 423–438, doi: 10.1145/2517349.2522737.

[7] B. Li, H. Zhang, and J. Wang, "Real-time Analytics with Apache Spark in Retail: Opportunities and Challenges," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 1943–1948, doi: 10.1109/BigData.2015.7363959.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *Commun. ACM*, vol. 52, no. 6, pp. 69–77, 2009, doi: 10.1145/1516046.1516061.

[9] A. Ghosh, P. Sarkar, and B. Basu, "Real-time System Monitoring Using Python: A Lightweight Monitoring Toolkit," *J. Comput. Sci. Eng.*, vol. 25, no. 2, pp. 35–42, 2021.

[10] M. A. Beyer and D. Laney, "The Importance of 'Big Data': A Definition," *Gartner Research*, Tech. Rep. G00235055, Jun. 2012. [Online]. Available: https://www.gartner.com/doc/2057415

[11] P. Raj and A. K. Saha, "Big Data Analytics with Spark: A Practitioner's Guide," in *Big Data Analytics*, Cham: Springer, 2015, pp. 169–193, doi: 10.1007/978-81-322-2647-5_9.

[12] R. Tolosana-Calasanz et al., "Real-Time Big Data Processing for Smart Retail," in *Proc. Int. Conf. Smart Cities*, 2018, pp. 58–63, doi: 10.1007/978-3-030-11464-0_8.

[13] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013, doi: 10.14778/2536222.2536229.

[14] M. Armbrust et al., "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2018, pp. 601–613, doi: 10.1145/3183713.3190664

Chapter 8: Appendix (Snapshots)

```
PS C:\Users\borut\Desktop\BDA\LAB_PROJECT> spark-submit spark_streaming.py
25/06/30 16:26:54 WARN Shell: Did not find winutils.exe: java.io.FileNotFoundException: java.io.FileNotFoundException: HADOOP_HOME and hadoop.home.dir are unset. -see https://wiki
apache.org/hadoop/WindowsProblems
25/06/30 16:26:55 INFO SparkContext: Running Spark version 3.5.5
25/06/30 16:26:55 INFO SparkContext: OS info Windows 11, 10.0, amd64
25/06/30 16:26:55 INFO SparkContext: Java version 17.0.12
25/06/30 16:26:55 INFO ResourceUtils: ==============================================================
25/06/30 16:26:55 INFO ResourceUtils: No custom resources configured for spark.driver.
25/06/30 16:26:55 INFO ResourceUtils: ==============================================================
25/06/30 16:26:55 INFO SparkContext: Submitted application: LiveInsightRetail
25/06/30 16:26:55 INFO ResourceProfile: Default ResourceProfile created, executor resources: Map(cores -> name: cores, amount: 1, script: , vendor: , memory -> name: memory, amoun
: 1024, script: , vendor: , offHeap -> name: offHeap, amount: 0, script: , vendor: ), task resources: Map(cpus -> name: cpus, amount: 1.0)
25/06/30 16:26:55 INFO ResourceProfile: Limiting resource is cpu
25/06/30 16:26:55 INFO ResourceProfileManager: Added ResourceProfile id: 0
25/06/30 16:26:55 INFO SecurityManager: Changing view acls to: Harsha
25/06/30 16:26:55 INFO SecurityManager: Changing modify acls to: Harsha
25/06/30 16:26:55 INFO SecurityManager: Changing view acls groups to:
25/06/30 16:26:55 INFO SecurityManager: Changing modify acls groups to:
25/06/30 16:26:55 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Harsha; groups with view permissions: EMPTY; users
with modify permissions: Harsha; groups with modify permissions: EMPTY
25/06/30 16:26:55 INFO Utils: Successfully started service 'sparkDriver' on port 54501.
25/06/30 16:26:55 INFO SparkEnv: Registering MapOutputTracker
25/06/30 16:26:56 INFO SparkEnv: Registering BlockManagerMaster
25/06/30 16:26:56 INFO BlockManagerMasterEndpoint: Using org.apache.spark.storage.DefaultTopologyMapper for getting topology information
25/06/30 16:26:56 INFO BlockManagerMasterEndpoint: BlockManagerMasterEndpoint up
25/06/30 16:26:56 INFO SparkEnv: Registering BlockManagerMasterHeartbeat
25/06/30 16:26:56 INFO DiskBlockManager: Created local directory at C:\Users\borut\AppData\Local\Temp\blockmgr-70fca24c-a2b0-4987-b554-92afd83d96f2
25/06/30 16:26:56 INFO MemoryStore: MemoryStore started with capacity 434.4 MiB
25/06/30 16:26:56 INFO SparkEnv: Registering OutputCommitCoordinator
25/06/30 16:26:56 INFO JettyUtils: Start Jetty 0.0.0.0:4040 for SparkUI
25/06/30 16:26:56 INFO Utils: Successfully started service 'SparkUI' on port 4040.
25/06/30 16:26:56 INFO Executor: Starting executor ID driver on host HARSHAVARDHAN
25/06/30 16:26:56 INFO Executor: OS info Windows 11, 10.0, amd64
25/06/30 16:26:56 INFO Executor: Java version 17.0.12
25/06/30 16:26:56 INFO Executor: Starting executor with user classpath (userClassPathFirst = false): ''
25/06/30 16:26:56 INFO Executor: Created or updated repl class loader org.apache.spark.util.MutableURLClassLoader@6b38f8a4 for default.
25/06/30 16:26:56 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 54502.
25/06/30 16:26:56 INFO NettyBlockTransferService: Server created on HARSHAVARDHAN:54502
25/06/30 16:26:56 INFO BlockManager: Using org.apache.spark.storage.RandomBlockReplicationPolicy for block replication policy
25/06/30 16:26:56 INFO BlockManagerMaster: Registering BlockManager BlockManagerId(driver, HARSHAVARDHAN, 54502, None)
25/06/30 16:26:56 INFO BlockManagerMasterEndpoint: Registering block manager HARSHAVARDHAN:54502 with 434.4 MiB RAM, BlockManagerId(driver, HARSHAVARDHAN, 54502, None)
25/06/30 16:26:56 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, HARSHAVARDHAN, 54502, None)
25/06/30 16:26:56 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, HARSHAVARDHAN, 54502, None)
C:\spark-3.5.5-bin-hadoop3\python\lib\pyspark.zip\pyspark\streaming\context.py:72: FutureWarning: DStream is deprecated as of Spark 3.4.0. Migrate to Structured Streaming.
[INFO] Streaming context started. Waiting for data...
[RECEIVED] TXN008772,26-05-2024 08:19,SmartRetail,Koramangala,4,5,1010876.0,Snacks,Lays Chips,4,20.0,80.0,7.29,72.71,0.0,UPI
[RECEIVED] TXN002294,26-05-2024 08:24,SmartRetail,Indiranagar,1,5,1038379.0,Snacks,Chocolate,5,40.0,200.0,3.65,196.35,1.0,Cash
[RECEIVED] TXN024029,26-05-2024 08:35,SmartRetail,Whitefield,2,5,1065581.0,Beverages,Bisleri Water (1L),1,20.0,20.0,1.56,18.44,0.0,UPI
[RECEIVED] TXN023311,26-05-2024 08:36,SmartRetail,Yelahanka,5,5,1065767.0,Beverages,Pepsi (2L),1,80.0,80.0,3.51,76.49,0.0,Card
```

Figure 8.1 Spark Runtime Snapshot *Spark Streaming continuously ingests and transforms retail data using a stateful MapReduce pipeline. This enables real-time decision-making by providing up-to-date analytics on branches, products, and payment patterns all directly derived from the live data stream.*

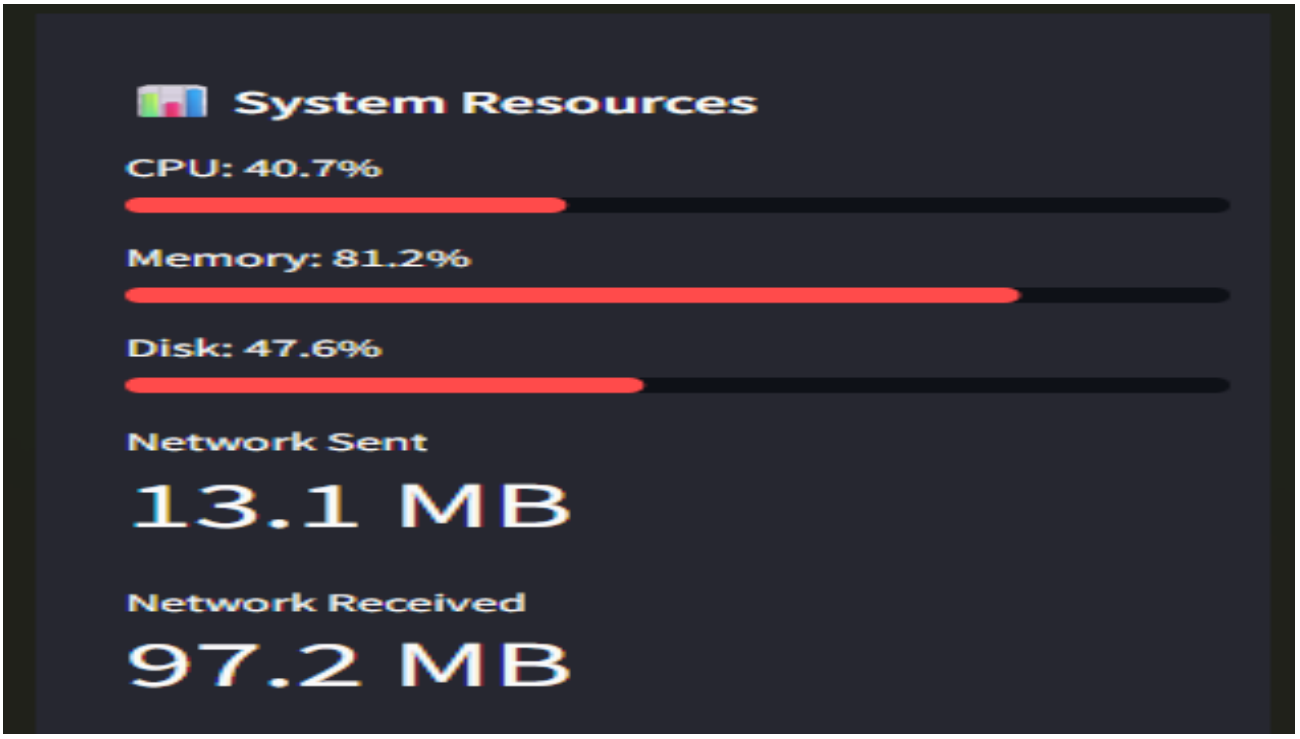

Figure 8.2 System Resources Overview. *Displays real-time resource usage, showing CPU at 40.7%, memory at 81.2%, and disk at 47.6%. Network statistics reveal 13.1 MB sent and 97.2 MB received, indicating significant data activity*

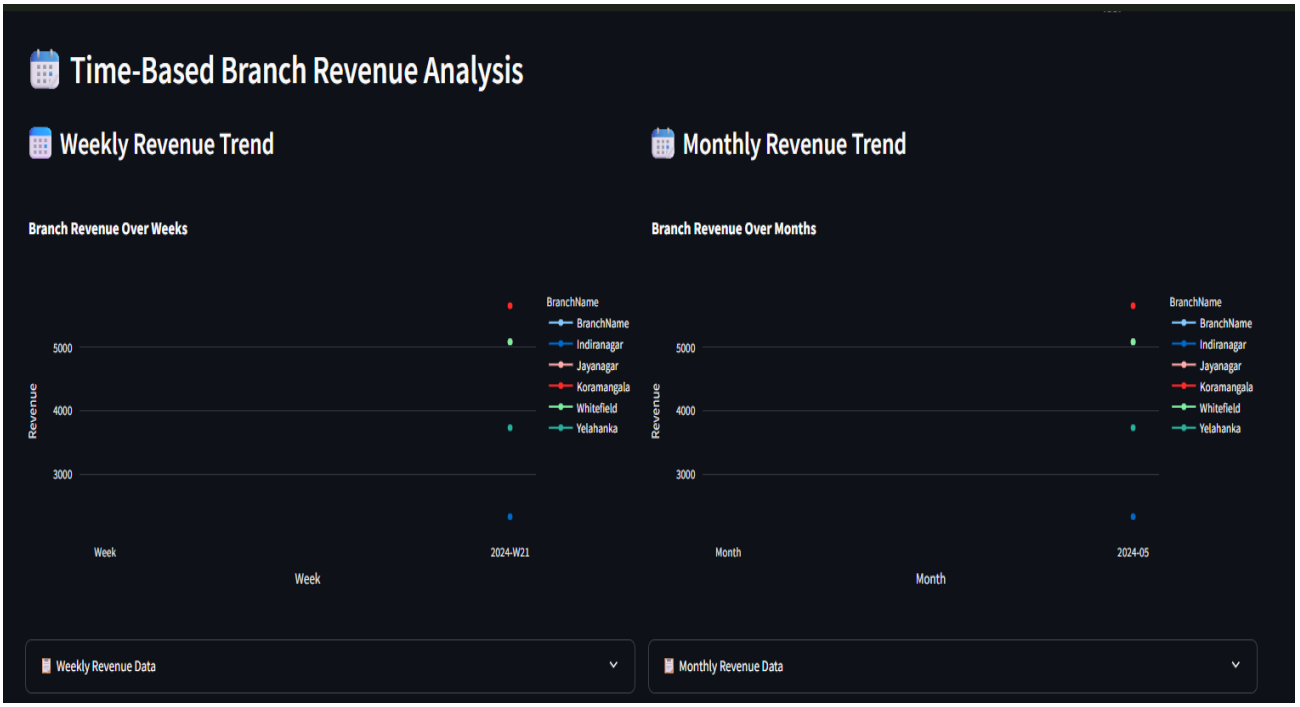

Figure 8.3 Time-Based Branch Revenue Analysis. *Shows weekly and monthly revenue trends for different branches*

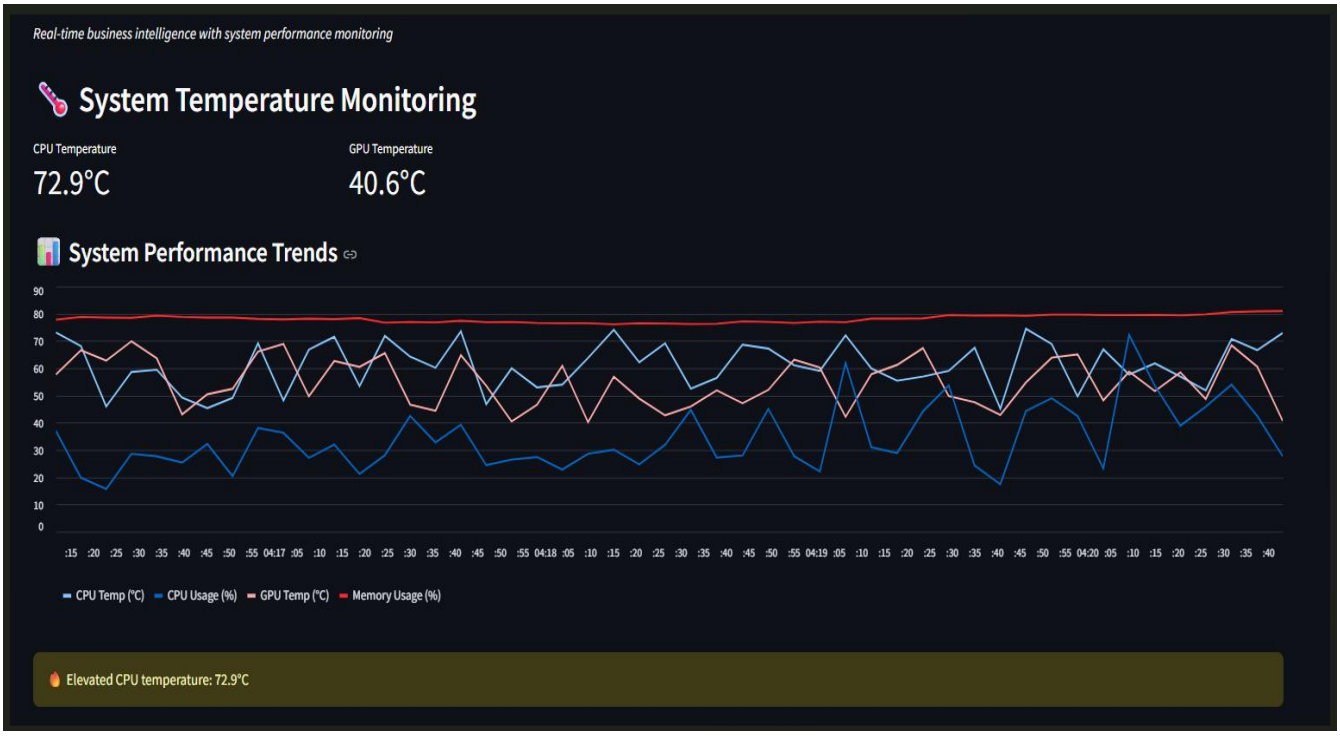

Figure 8.4 System Temperature Monitoring .*Tracks CPU and GPU temperature trends during processing, indicating elevated CPU temperature at 72.9°C.*

***Fig 8.5 Key Metrics** . Summarizes essential business data including total revenue (₹26,561), units sold, and active branches.*