Operational Analytics: A Log-Driven Dashboard

Team Dataminers

Abhitosh
abhitosh2024@iisc.ac.in

Amit Nitin Joshi
amitj@iisc.ac.in

Naik Raghavendra Narottam
nraghavendra@iisc.ac.in

Vignesh S.
vigneshs@iisc.ac.in

1. Problem Definition

In modern businesses, software systems generate vast amounts of logs, capturing information about system performance, application errors, security events and user behavior. However, they struggle to effectively process and analyze this log data for actionable insights. The problem resolves around the difficulty of making sense of these vast, often unstructured log data streams to ensure timely and effective decision-making. Without such an infrastructure, the operational teams in the organization struggle to identify trends, diagnose issues, predict failures, and optimize processes. Traditional methods of analyzing log data, such as manual review of raw logs or using static reporting tools, require significant manual effort, are time-consuming and heavily inefficient. Without effective monitoring and analysis, potential issues may go unnoticed until they escalate into significant outages or performance degradations, leading to costly downtime and decreased user satisfaction.

1.1 Motivation

A log-driven infrastructure enables organizations to transform log data from systems, applications, and devices into valuable, actionable insights. The main motivation behind adopting this infrastructure is to provide real-time monitoring and analysis, allowing organizations to detect and address issues quickly, reducing downtime and improving system reliability. Logs capture critical data on performance, errors and events which can be analyzed proactively to identify anomalies, predict failures, and resolve challenges before they escalate. This infrastructure shall support data-driven decision-making by offering insights into system behavior, which helps optimize workflows., By centralizing and automating log management, organizations can handle increasing data volumes efficiently as they scale.

1.2 Design Goals and Features

Distributed data parallel processing must be enabled the efficient handling of large-scale log data by splitting tasks across multiple nodes for concurrent analysis. It shall speed up data processing, ensuring real-time monitoring and faster issue detection, even during high data volumes. For log streams, it shall distribute workloads across nodes. This method shall enhance fault tolerance by allowing the system to continue functioning if a node fails, with redundancy and data replication preventing data loss. It also optimizes resource allocation through load balancing, dynamically assigning tasks to the most available nodes. Overall, distributed processing improves scalability, reliability, and efficiency, enabling organizations to manage and analyze vast amounts of log data quickly and accurately for better decision-making and system monitoring.

Visualization of system KPIs derived from application logs involves presenting key metrics like failure rates, fault rates, and system load through interactive dashboards. These KPIs help track system performance and find issues such as errors, malfunctions, or resource bottlenecks. By visualizing these metrics in real-time using charts, graphs, and heatmaps, teams can quickly detect anomalies and make informed decisions. This approach enhances operational monitoring, enabling quicker responses to issues, improving system stability, and optimizing performance.

Lastly, it must offer scalability, flexibility, and high-performance analytics so that it can enable Big Data platforms. This will extract valuable insights, drive data-driven decisions, and improve operational efficiency.

2. Approach and Methods

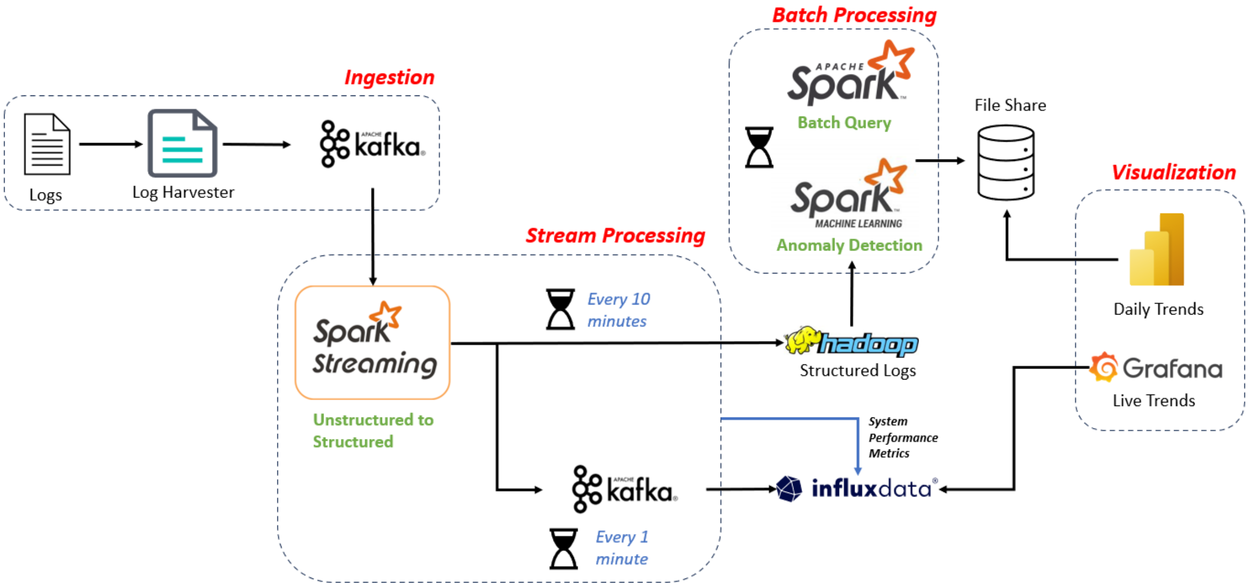

Fig 1. Log processing pipeline architecture.

We have created an end-to-end pipeline shown in Fig 1. designed to ingest, process, and visualize log data. The system can transform unstructured log data into a structured format, detecting anomalies in system performance, and providing real-time and historical insights through visualizations. This design employs multiple data processing frameworks and technologies such as Apache Kafka, Apache Spark, Hadoop, InfluxDB, PowerBI and Grafana, offering both batch and stream processing capabilities to ensure robust data analysis and monitoring. It combines stream and batch processing using Apache Kafka and Apache Spark, allowing for real-time and batch-level data transformation and analysis. The use of InfluxDB and Grafana for visualization ensures that insights into system performance are readily available, both for immediate action (live trends) and long-term analysis (daily trends). By using these technologies, the system can offer prompt and actionable insights, improving the ability to monitor and optimize system performance.

2.1 Ingestion

The ingestion process begins with raw log data that is generated by the system. This is collected by a log harvester, like fluent bit, filebeat or even a custom implementation. The log harvester spools the log files and pushes the log data to Apache Kafka for later processing.

Apache Kafka serves as the backbone of this architecture, acting as a high-throughput messaging system that ensures the reliable transport of log data between different components of the system. Kafka allows for scalability, fault tolerance, and reliability. It also decouples the producers of the log data from the consumers, enabling easier management of data flow.

Additionally, the historical system logs can be imported into HDFS, thereby providing access to historical and real-time system logs for analytics operations.

2.2 Structured Logs

The raw logs ingested by the system, either through Kafka streams or historical logs files present in the HDFS, are converted to structured logs. Here, we use Drain3[2,4,7] which is a log template miner that can extract templates from a stream of log messages.

Drain3[2,4,7] utilizes a structured process for analyzing log data, starting with tokenization which breaks down log messages into individual tokens. For example, logs like "Connection from 182.168.1.1 on port 80" are tokenized into components such as "Connection", "from" and "192.168.1.1".

Then a tree is built where each token is a node, and common tokens are grouped together. This structure reveals

similarities and helps analyze the relationships between different log entries.
Pattern identification is central to Drain3's[2,4,7] functionality where it finds recurring patterns like "Connection from [IP] on port [PORT]". These patterns are then extracted for creating structured log formats.

Example schema of the structured logs produced by Drain3[2,4,7] is depicted in Table 1.

Column Name	Data-Type	Description
Date	String	The date extracted from the log message.
Time	String	The time extracted from the log message.
ms	Integer	Milliseconds extracted from the log message.
Level	String	Log severity level (e.g., INFO, ERROR, WARN).
Component	String	Log component/module generating the message.
Content	String	The dynamic content of the log message.
EventTemplate	String	From "Content", data, Template representing the static structure of the log message.
EventId	Integer	A unique identifier assigned to each log template by Drain Parser.

Table 1 – Structured log schema produced by Drain3[2,4,7] for an application log.

2.3 Stream Processing

Once the data is ingested into Kafka, it is processed in real-time using Apache Spark Streaming and Apache Spark Dataframes. Spark Streaming handles converting unstructured log data into a structured format suitable for analysis. This transformation from unstructured to structured data is done using Drain3[2,4,7] on the ingested batch of logs. This produces a Spark dataframe with schema depicted in *Table 1.*
Stream processing is performed at two time intervals: every minute and every 10 minutes, using a batch length window query. The Spark dataframe is then provided for downstream processing. At the same time, it is persisted into HDFS along with structured historical data for batch processing later.

2.4 Batch Processing

In addition to the stream processing layer, batch processing is employed to perform more complex operations on accumulated data. The system uses Apache Spark's batch query functionality to process data in larger chunks. Processed data from batch queries is stored in Hadoop's HDFS, allowing for scalable and distributed storage of structured logs. The HDFS system ensures that large volumes of data can be stored and accessed efficiently, supporting the analytics and visualization needs of the system.
In the project, we perform batch queries on structured HDFSv2[1,6] log data to produce the reports mentioned in *Table 2*. These reports are saved in HDFS.

Report	Description
Log Severity	Aggregates count by log level (ERROR, WARN, INFO, FATAL).
Exceptions	Counts occurrences of specific exceptions and orders them by date.
Write Latency	Calculates the average write latency for slow block receiver events.
Speed Report	Calculates the transfer speed for HDFS write operations
Monthly Severity	Groups number of log entries by log level and by months, counts distinct event ids.
Retry Count	Counts the number of retries for server connections and orders them by count.

Table 2: Various reports generated on HDFSv2[1,6] structured logs by batch processing.

2.5 Anomaly Detection

Spark's machine learning capabilities are used for anomaly detection. This is used to find unusual patterns in system performance, which can show potential issues or failures. The anomaly detection model helps check system health and show areas that require attention, ensuring the robustness of the system.

Our anomaly detection algorithm is inspired by Loglizer[3,8] and experiments were performed on the HDFSv2[1,6] log dataset. Anomaly detection is performed on each block ID provided by the HDFSv2[1,6] logs. Such a mechanism can be used to show data integrity issues, hardware failures, and resource imbalances. It can aid in detecting abnormal access patterns or inefficient block distribution.

We first group all structured logs using block ID. Then, we calculate event sequence using the EventIDs.

We then use TF-IDF, which is a technique used to assess the relevance of a term within a document relative to its frequency across multiple documents. TF-IDF is applied to the event sequence of EventIDs even by first tokenizing it into individual terms. The HashingTF step computes the term frequency for each token in the sequence, creating a sparse feature vector. Then, the IDF (Inverse Document Frequency) model adjusts these term frequencies by considering how common or rare each term is across the entire dataset, giving higher weight to rare terms. The result is a transformed feature vector being the event sequence with weighted terms based on their importance. This feature vector is used to train a logistic regression model on the structured HDFSv2[1,6] logs.

The batch processing system can then use this anomaly detection model to generate reports on detected anomaly counts for a given time span.

2.6 Visualization

For real-time monitoring and historical analysis, the system integrates InfluxDB and Grafana for visualization. InfluxDB is used to store time-series data, such as system performance metrics, that is captured every minute. These metrics, including CPU usage, memory usage, and other system health indicators, are crucial for monitoring system performance in real time.

Grafana serves as the visualization tool for both live and historical trends. It connects to InfluxDB to fetch performance metrics and display them on dashboards. Grafana provides an intuitive interface for visualizing real-time data (live trends) and historical data (daily trends), making it easier for users to interpret the system's behavior and take corrective actions when necessary.

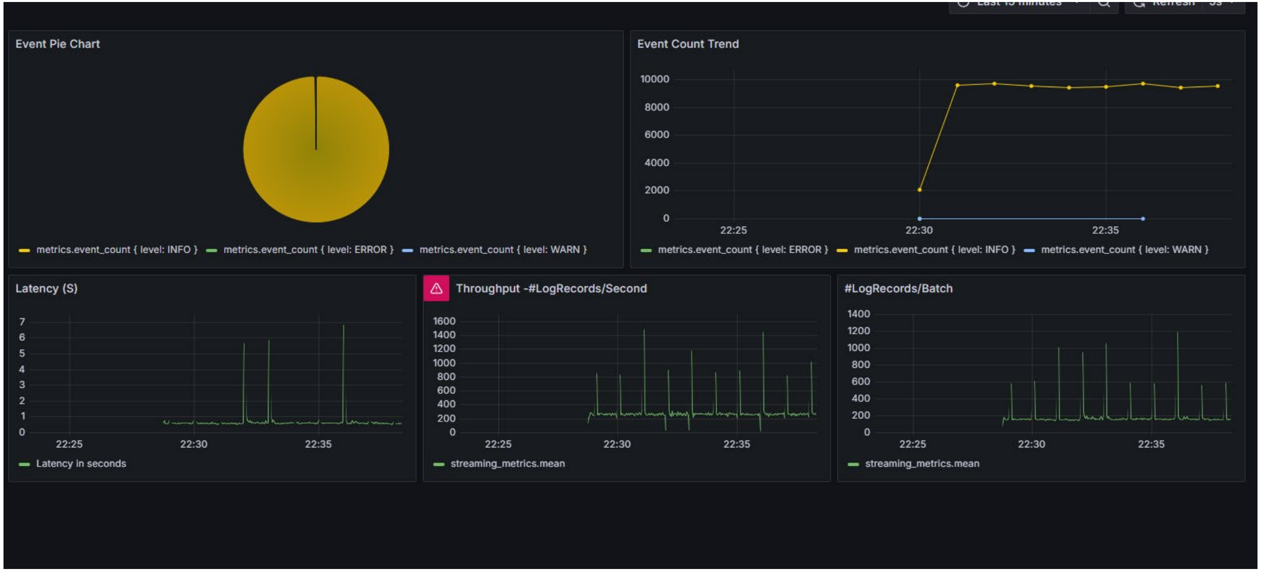

Fig 2. Visualization of real-time HDFSv2[1,6] logs and logging system metrics in a Grafana dashboard. Once the stream processing is performed, the results are published to InfluxDB. Grafana queries InfluxDB for above visualization.

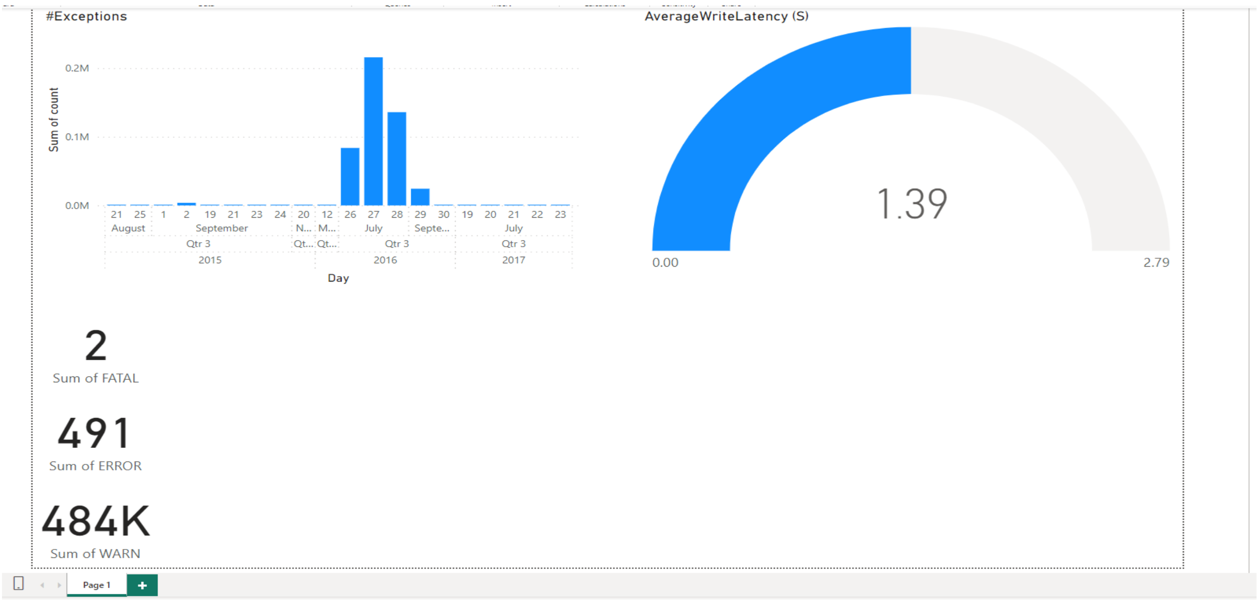

Fig 3. Generated reports from Batch processing visualized in PowerBI.

2.7. Experiments with Cloud Deployment

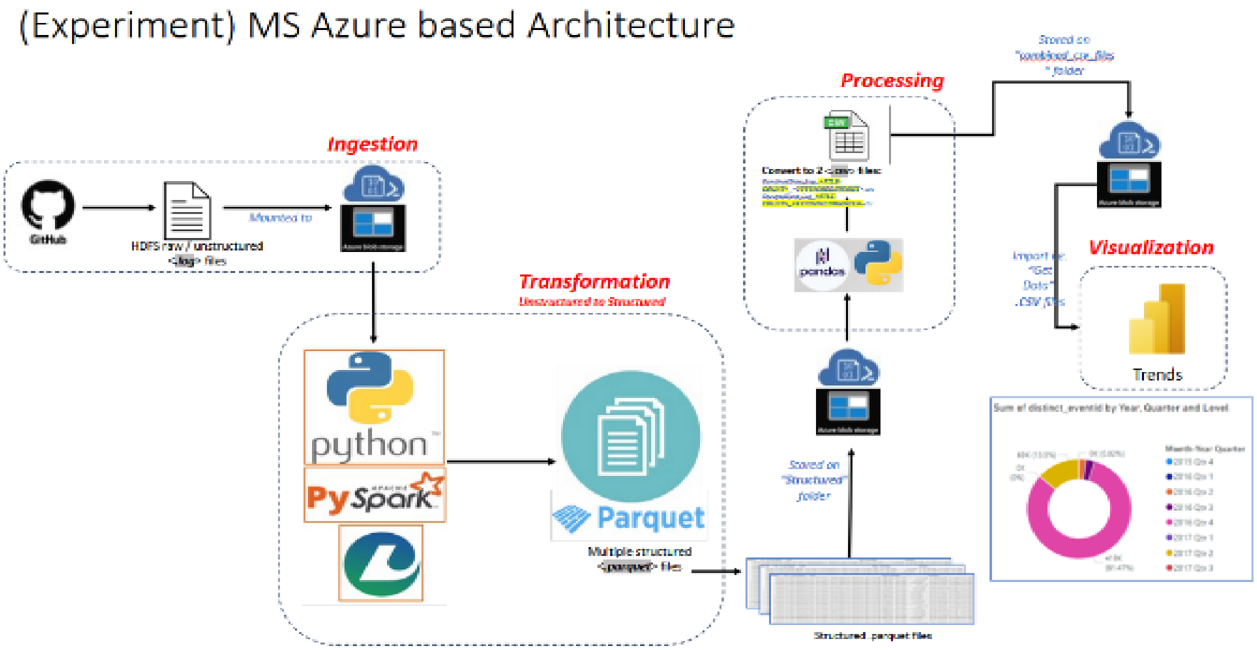

Fig. 4: Experimental Architecture with Azure Databricks on Cloud

Architecture

The architecture integrates Azure Blob Storage, Azure Databricks, Spark/PySpark, Azure Data Factory, and Power BI for log processing. Raw logs are ingested into Azure Blob Storage, parsed into structured data using Drain Parser in Databricks, and stored as Parquet files. Azure Databricks performs transformations and aggregations using Spark, while Azure Data Factory orchestrates the pipeline workflow. Processed data is exported as CSV files to Blob Storage, enabling Power BI to visualize trends and insights. This is exhibited in *Fig. 6*.

3. Evaluation

3.1 Experiment Design

For evaluation, a HP ZBook G6 laptop with i9 11th Gen, 64GB RAM, 12 virtual processors, 4GB NVidia RTX A2000, 1TB SSD, Windows 10 22H2 was used. Here, a WSL instance of Ubuntu-22.04 was created with 16GB RAM and 6 virtual cores. All pre-requisite software – Apache Hadoop, Apache Spark, PySpark, Apache Kafka, InfluxDB and Grafana was setup within the WSL instance. All these components are running in standalone mode, which is the minimum configuration for this project.
For evaluation, we have used HDFSv2[1,6] log dataset from LogPai.

3.2 Model Evaluation

The anomaly detection model was trained on 100, 000 log entries from HDFSv2[1,6], along with a dataset that classified the block IDs from these log entries as anomalies. Table 3 describes the training metrics of training as well as model performance metrics.

Name	**Value**
Training duration	106.27 s
Accuracy	0.7723
Precision	0.9714
Recall	0.5151
F1 Score	0.6733
ROC-AUC	0.7723

Table 3: Training metrics.

The model demonstrates high precision, indicating that most predicted anomalies are correct, but its recall is relatively low, meaning it misses many actual anomalies. The F1 score suggests a moderate balance between precision and recall, while the ROC-AUC indicates reasonable overall performance.
During inference, the model is relatively fast and can process a million log lines for anomalies in less than 10 seconds.

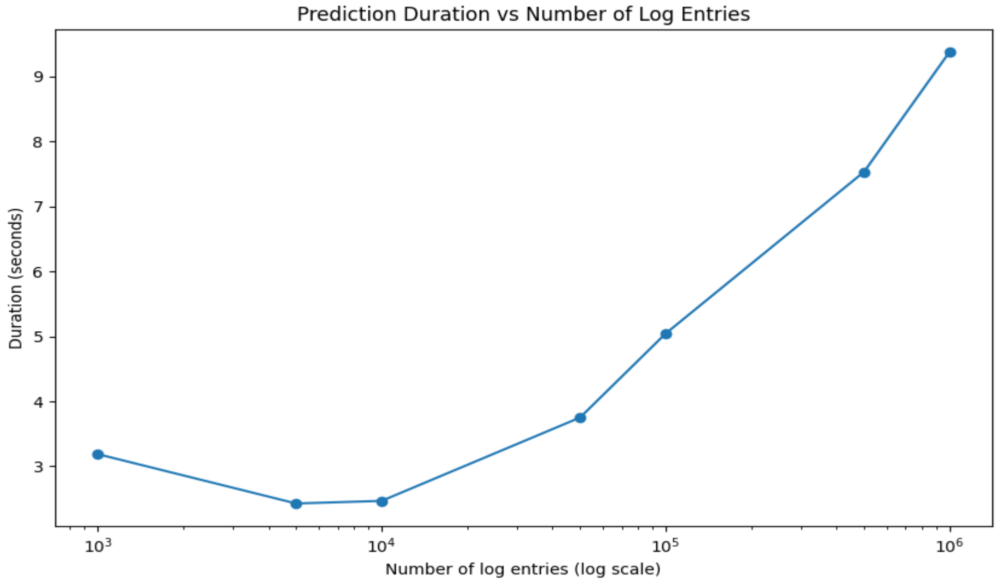

Fig 4. Plot of Anomaly detection duration vs number of log lines on a logarithmic scale. The evaluation was performed was HDFSv2[1,6] dataset.

3.3 Evaluation of latency and throughput

We evaluated the performance of the system by gradually increasing the input rate of feeding raw logs and measuring the latency and throughput. We measure latency as the time to convert unstructured logs to structured dataframe within spark structured stream processing job.

Experiment setup

For the experiment, we increased the rate of feeding raw logs from 10 to 2000 lines of logs per second in 15 minutes interval. Fig 2 shows the results of the evaluation, as measured by metrics introduced in stream processing pipeline.

Observations

- Latency did not change significantly with increasing input rate.
- Throughput increased almost linearly with input rate before saturating at about 1500.

Analysis

- Unstructured to structured conversion contributes to the latency of the system.
- Throughput of the system was limited by the latency of the Kafka stream processing.
- The system scales well with increase in input stream rate until a certain point limited by streaming throughput of Kafka input stream.
- Latency of the system is limited by the performance of unstructured to structured conversion process.

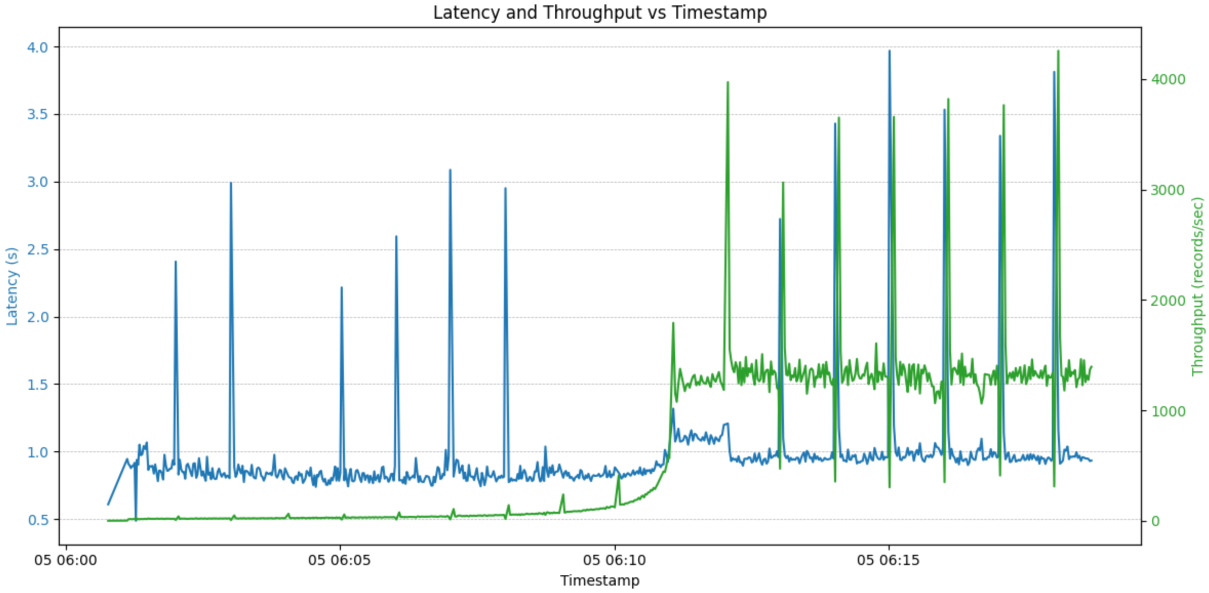

Fig 5. Plot of latency and throughput vs timestamp. In the experiment, the input rate of feeding raw log lines was increased from 10 to 2000 in 15 minutes interval.

5. Summary

The architecture presented provides a comprehensive solution for managing and visualizing log data in a scalable and efficient manner. It combines stream and batch processing using Apache Kafka and Apache Spark, allowing for real-time and batch-level data transformation and analysis. The use of InfluxDB and Grafana for visualization ensures that insights into system performance are readily available, both for immediate action (live trends) and long-term analysis (daily trends). By leveraging these technologies, the system can offer timely and actionable insights, improving the ability to monitor and optimize system performance.

6. References

1. Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, Michael R. Lyu. "Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics." ("GitHub - logpai/loghub: A large collection of system log datasets for ...") ("GitHub - logpai/loghub: A large collection of system log datasets for ...") IEEE International Symposium on Software Reliability Engineering (ISSRE), 2023.

2. Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. ("LogAnMeta: Log Anomaly Detection Using Meta Learning - ResearchGate") "Drain: An Online Log Parsing Approach with Fixed Depth Tree, Proceedings of the 24th International Conference on Web Services (ICWS), 2017." ("logparser/logparser/Drain/README.md at main - GitHub")

3. Shilin He, Jieming Zhu, Pinjia He, Michael R. Lyu. Experience Report: System Log Analysis for Anomaly Detection, *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2016. ("GitHub - logpai/loghub: A large collection of system log datasets for ...") 中文版本 *ISSRE Most Influential Paper*

4. Use open source Drain3 log-template mining project to monitor for network outages

5. Fingerprinting the Datacenter: Automated Classification of Performance Crises, by Peter Bodík, Moises Goldszmidt, Armando Fox, Hans Andersen. *Microsoft*

6. loghub

7. Drain3

8. loglizer