Real-Time Log Analysis Using Hadoop and Spark

Table of Contents

1. Abstract

Modern organizations face challenges with **massive, fast, and diverse log data** that conventional analytic tools can't handle. This project, "Real-Time Log Analysis Using Hadoop and Spark," creates a distributed pipeline for **collecting, ingesting, processing, and aggregating logs** using **Apache Kafka/Flume, Hadoop HDFS, and Apache Spark Streaming**. The system performs **real-time trend detection, anomaly alerting, and generates actionable operational dashboards** through scalable computation and storage, providing a foundation for continuous monitoring and rapid incident response.

---

2. Introduction

Log data is crucial for understanding system behavior and health. However, the **increasing scale and velocity of log streams** overwhelm traditional batch analytics, causing delays in detecting incidents, security threats, and operational issues. This project uses **Hadoop's scalable storage** and **Spark's real-time streaming engine** to build a flexible, high-performance pipeline for live log analytics. The solution provides **low-latency insights, early warnings, and historical reporting**, enabling proactive infrastructure and user experience management.

---

3. Problem Statement

Traditional logging infrastructures face several key challenges:

- **Scalability**: Inability to manage thousands or millions of log events per second from various sources.
- **Latency**: Slow ETL and batch jobs cause significant delays in critical incident detection.
- **Diversity**: Logs from applications, servers, and devices have differing formats and semantics.
- **Operational Insight**: Difficulty in real-time identification of trends, spikes, and anomalies across distributed systems.
- **Storage & Query**: The need for both immediate streaming analytics and historical batch queries over long periods.

The project aims to address these issues by designing a **distributed system** with real-time streaming, scalable storage, and live analysis and alerting capabilities.

---

4. Literature Review

Industry and academic literature highlights the necessity for **distributed, stream-based analytics** in operational monitoring:

- **Distributed Log Ingestion**: **Kafka and Flume** are recognized for high-throughput, fault-tolerant log streaming.
- **Stream Analytics**: **Apache Spark Streaming** and similar frameworks enable micro-batch, continuous computation as data arrives.
- **Operational Visualization**: **Elasticsearch, Kibana, and Grafana** are standard tools for live dashboards and exploratory data analysis.
- **Research Findings**: Studies confirm that **windowed aggregations, alert thresholds, and automated anomaly detection** significantly improve the speed of detecting and remediating outages, attacks, or SLA breaches compared to batch architectures.

---

5. Architecture & Technology Stack

System Design

Layer	Technology	Role
Ingestion	Kafka / Flume	Distributed real-time log shipping
Storage	HDFS, Hive	Durable, scalable log storage
Processing	Spark Structured Streaming	Real-time parsing, filtering, aggregation
Visualization	Grafana, Kibana	Dashboards and trend graphs
Machine Learning	Spark MLlib (optional)	Anomaly detection, event clustering

System Flow

1. **Log Collection**: Logs from web, application, and system sources are ingested in real-time via **Kafka or Flume**.
2. **Distributed Storage**: All logs are initially stored in **Hadoop HDFS**, partitioned by timestamp.
3. **Stream Processing**: **Spark Streaming** consumes logs from Kafka, then parses, filters, and aggregates data in sliding windows to support operational metrics and anomaly detection.

4. **Result Storage**: Aggregated data is saved to **Hive** for ad hoc SQL queries, with summaries optionally sent to **Elasticsearch** for live visualization.
5. **Visualization**: Real-time and historical dashboards are built using **Hive, Elasticsearch**, or other tools.
6. **Alerting**: Thresholds and anomaly detection algorithms trigger alerts for incident response.

---

6. Dataset Description

Data Sources

- **Web Server Logs**: Include fields such as timestamp, IP, URL, HTTP method, status code, and user agent.
- **Application Logs**: Contain transaction events, error messages, and custom activity.
- **System Logs**: Provide CPU/memory/disk metrics and syslog information.

Sample Schema

timestamp	ip	method	url	status	user_agent
2025-07-19 10:00:00	192.168.1.1	GET	/home	200	Mozilla/5.0
...	...	...	...	...	...

Logs are ingested as either text or CSV/JSON and then parsed into this standardized structure for subsequent analysis.

---

7. Detailed Algorithm & System Flow

Summarized System Steps

1. **Data Collection**
    - Identify log sources (web, app, system).
    - Configure **Kafka/Flume** for real-time collection and shipping to central topics/channels.

2. **Ingestion in Spark**
    - A **Spark Structured Streaming job** subscribes to the Kafka/Flume stream.
    - Logs are read in small intervals (e.g., every 5–10 seconds).
    - Raw logs are parsed into structured fields using regex or schema mapping.
3. **Processing & Transformation**
    - Logs are filtered by error conditions (e.g., status >= 400 or ERROR keyword).
    - Fields are extracted and enriched with context (e.g., GeoIP, user-agent analysis).
    - Data is aggregated with windowing (e.g., 10-minute sliding window) to compute metrics like request count, error rate, and top URLs/IPs.
4. **Analysis & Anomaly Detection**
    - Current metrics are compared to historical averages to identify spikes or anomalies.
    - (Optional) **Spark MLlib** can be used for unsupervised detection of unusual patterns.
5. **Persistence**
    - Both raw and processed logs are stored in **HDFS**.
    - Aggregates are saved to **Hive tables**, partitioned by time, for SQL analytics.
6. **Visualization & Reporting**
    - Dashboards query **Hive, Elasticsearch**, or direct Spark outputs for real-time/top-N metrics, error rates, and usage trends.
    - Alerts are triggered if error rates or traffic exceed defined thresholds.

Main Outputs

- Aggregated metrics (minute/hour/day)
- Lists of top URLs/IPs
- Anomaly and alert logs
- Visual and exported reports

---

8. Implementation Approach & Code Snippets

Environment Setup

```python
# Install Java, Spark, findspark
!apt-get install openjdk-11-jdk-headless -qq > /dev/null
!wget -q https://archive.apache.org/dist/spark/spark-3.5.0/spark-3.5.0-bin-hadoop3.tgz
!tar xf spark-3.5.0-bin-hadoop3.tgz
```

```
!pip install -q findspark

import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-11-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.5.0-bin-hadoop3"
import findspark; findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").appName("LogAnalysis").getOrCreate()
```

Load and Prepare Data

Python

```
from google.colab import files
uploaded = files.upload() # Upload 'sample_web_logs_fixed.csv'

df = spark.read.option("header", True).option("inferSchema", True).csv("sample_web_logs_fixed.csv")
from pyspark.sql.functions import hour, to_timestamp
df_with_time = df.withColumn("hour", hour(to_timestamp("timestamp")))
```

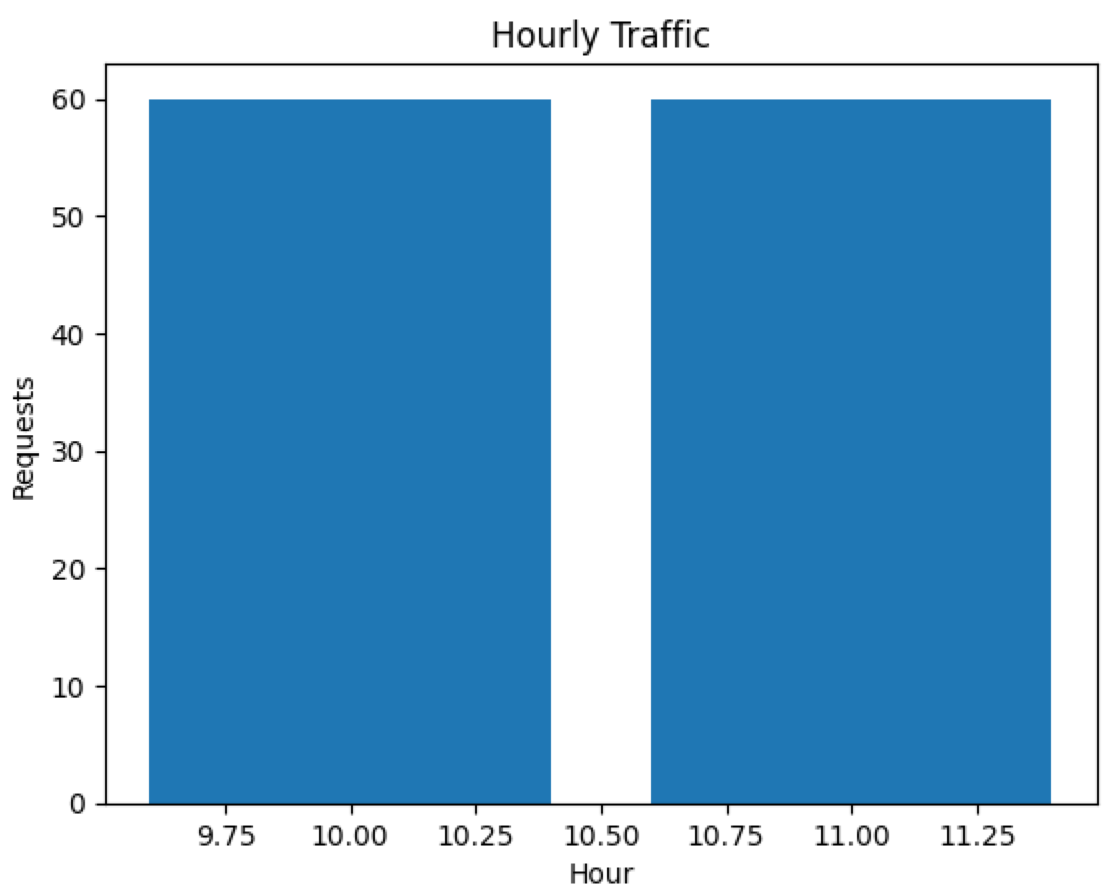
Aggregation Examples

Python

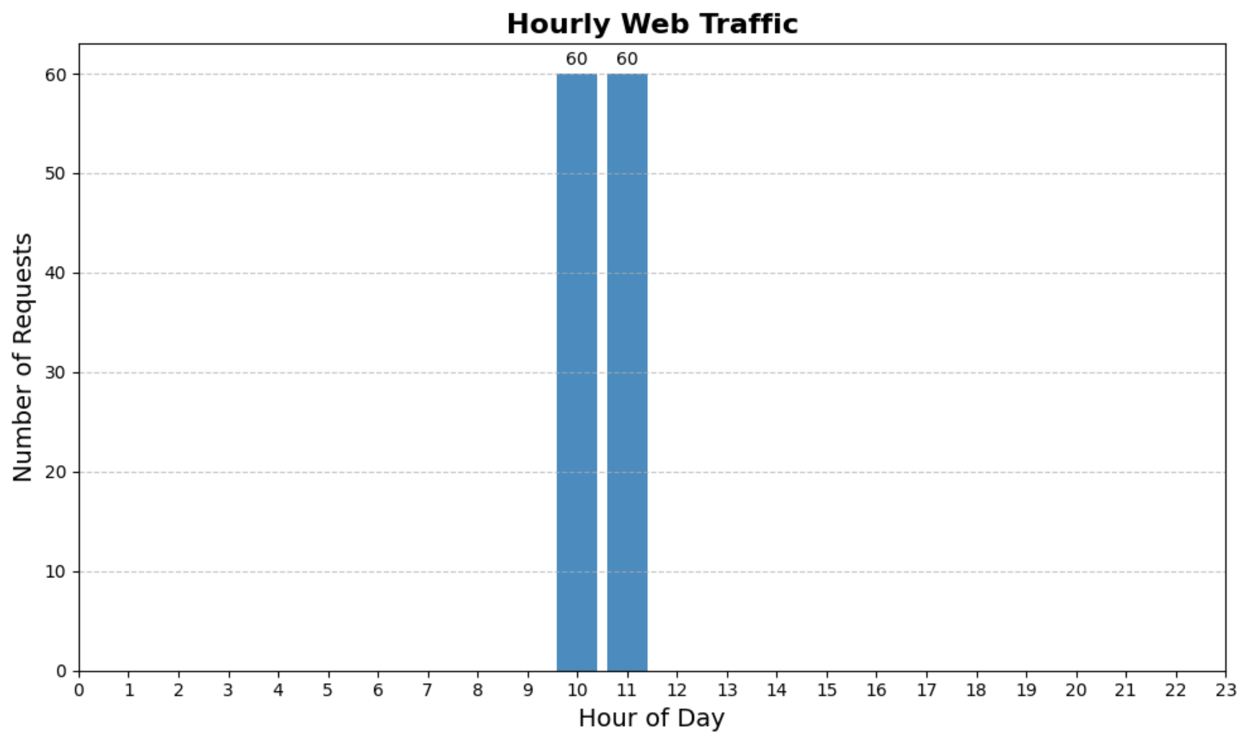
```
# Hourly traffic counts
traffic = df_with_time.groupBy("hour").count().orderBy("hour").toPandas()

# Top status codes
status_counts = df.groupBy("status").count().orderBy("status").toPandas()

# Top URLs
top_urls = df.groupBy("url").count().orderBy("count", ascending=False).limit(10).toPandas()

# Error rate by hour
from pyspark.sql.functions import col, count, lit, coalesce
```

```
error_requests_per_hour = df_with_time.filter(col('status') >= 400).groupBy('hour').count()
total_requests_per_hour = df_with_time.groupBy('hour').count()
error_rate_by_hour = error_requests_per_hour.join(
    total_requests_per_hour, 'hour', 'right'
).withColumn(
    'error_rate', (coalesce(error_requests_per_hour['count'], lit(0)) / total_requests_per_hour['count'])
).orderBy('hour')
```

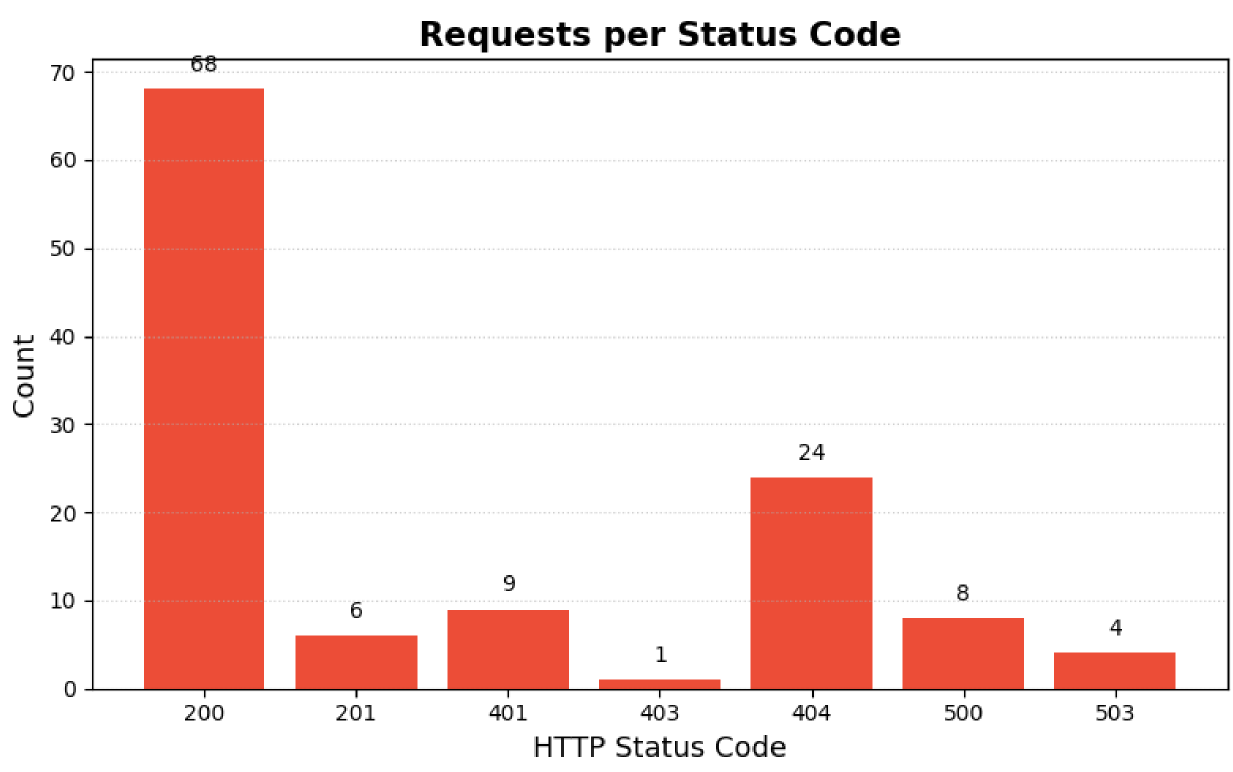
```
import matplotlib.pyplot as plt

traffic = df_with_time.groupBy("hour").count().orderBy("hour").toPandas()
plt.bar(traffic['hour'], traffic['count'])
plt.xlabel("Hour")
plt.ylabel("Requests")
plt.title("Hourly Traffic")
plt.show()
```



```
import matplotlib.pyplot as plt
import numpy as np

# Suppose traffic['hour'] is int 0-23
fig, ax = plt.subplots(figsize=(10,6))
bars = ax.bar(traffic['hour'], traffic['count'], color="#4b8bbe")

# Set x-axis as all 0-23 to show gaps as zeros if any hours are missing in
the data
ax.set_xticks(np.arange(0, 24, 1))
ax.set_xticklabels(np.arange(0, 24, 1))
ax.set_xlabel("Hour of Day", fontsize=14)
ax.set_ylabel("Number of Requests", fontsize=14)
ax.set_title("Hourly Web Traffic", fontsize=16, fontweight='bold')
ax.yaxis.grid(True, linestyle='--', alpha=0.7)

# Annotate bars
for bar in bars:
    height = bar.get_height()
    if height > 0:
        ax.annotate('{}'.format(height),
                    xy=(bar.get_x() + bar.get_width() / 2, height),
                    xytext=(0,3), # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom', fontsize=10)

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



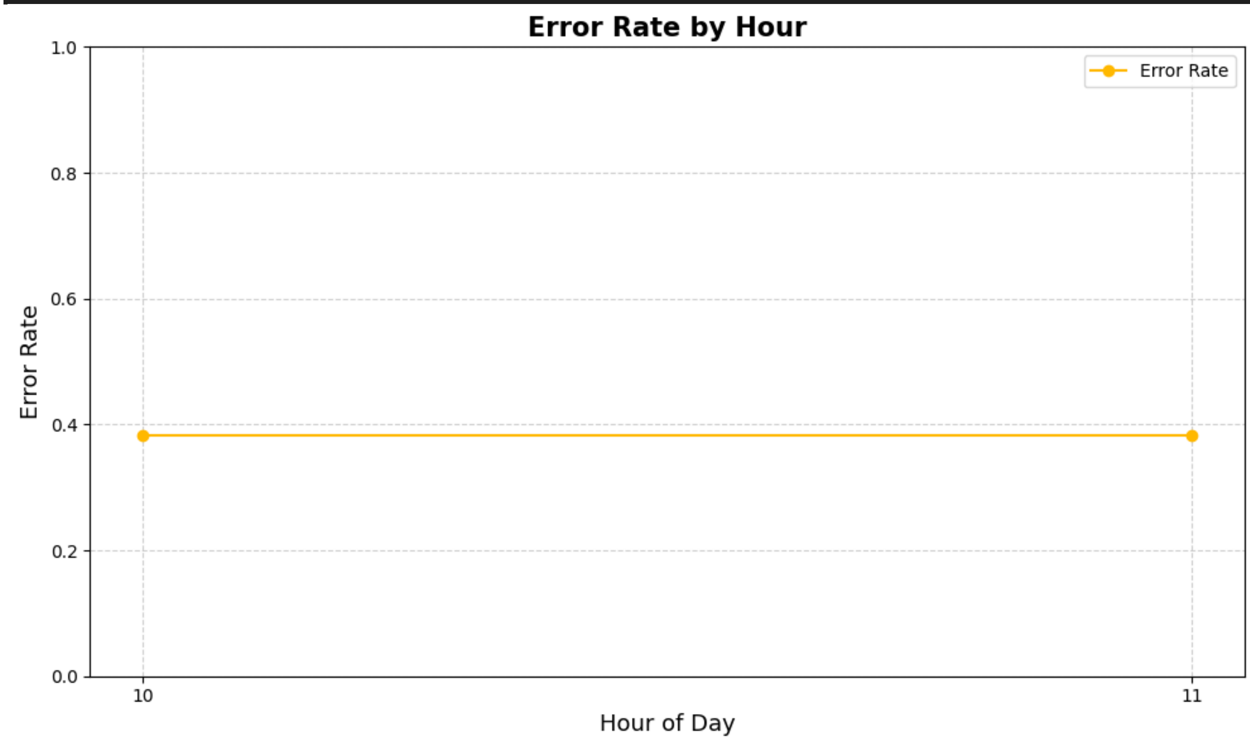

```
status_counts = df.groupBy("status").count().orderBy("status").toPandas()
plt.figure(figsize=(8,5))
plt.bar(status_counts['status'].astype(str), status_counts['count'],
color="#ec4d37")
plt.xlabel("HTTP Status Code", fontsize=13)
plt.ylabel("Count", fontsize=13)
plt.title("Requests per Status Code", fontsize=15, fontweight='bold')
plt.grid(axis='y', linestyle=':', alpha=0.6)
for idx, val in enumerate(status_counts['count']):
    plt.text(idx, val+2, str(val), ha='center', fontsize=10)
plt.tight_layout()
plt.show()
```



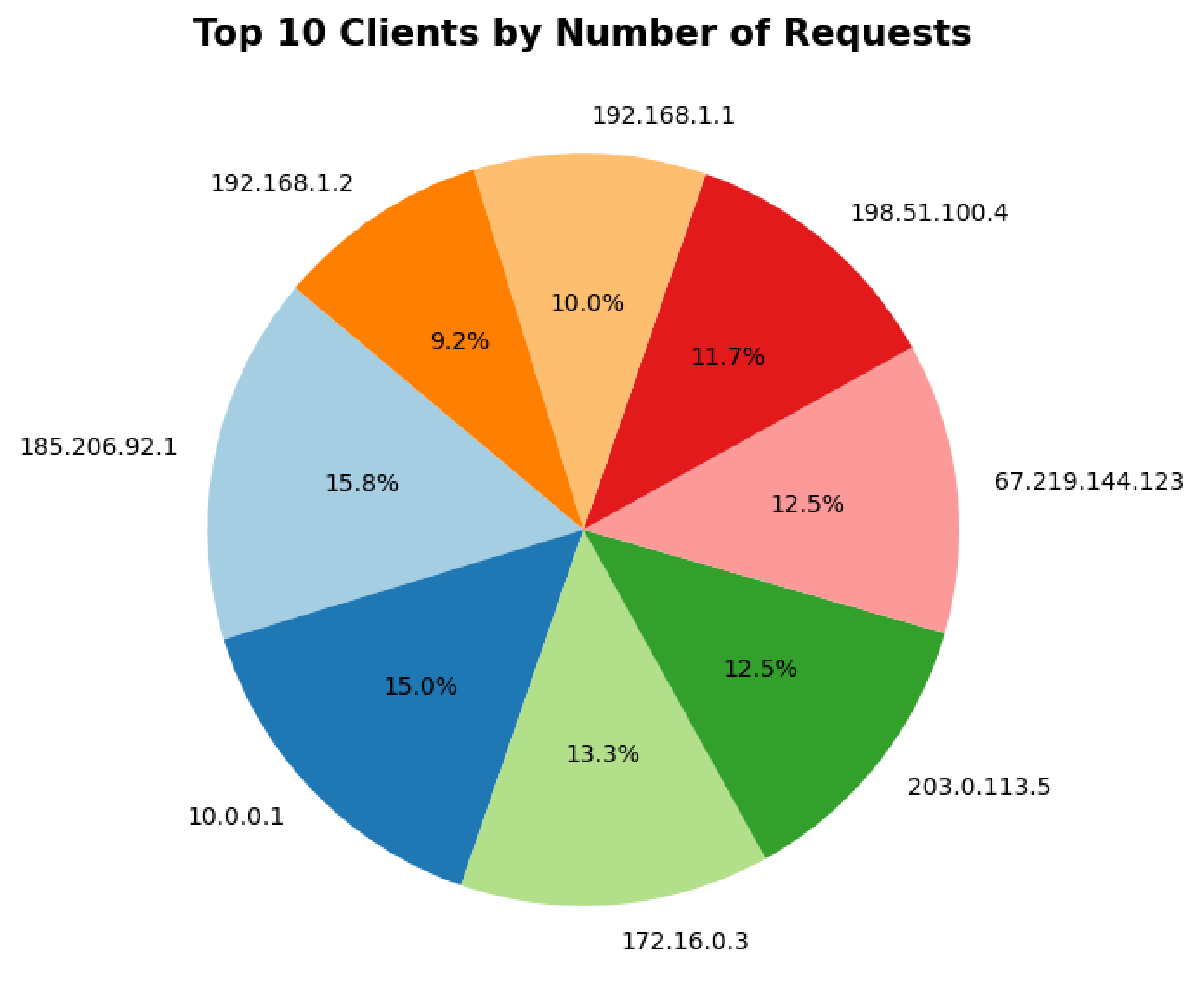

```
top_urls = df.groupBy("url").count().orderBy("count",
ascending=False).limit(10).toPandas()
plt.figure(figsize=(10,6))
bars = plt.barh(top_urls['url'], top_urls['count'], color="#64cc85")
plt.xlabel("Requests", fontsize=13)
plt.ylabel("URL", fontsize=13)
plt.title("Top 10 Requested URLs", fontsize=15, fontweight='bold')
plt.gca().invert_yaxis()
for bar in bars:
    plt.text(bar.get_width()+2, bar.get_y()+bar.get_height()/2,
str(int(bar.get_width())), va='center')
plt.tight_layout()
plt.show()

top_urls = df.groupBy("url").count().orderBy("count",
ascending=False).limit(10).toPandas()
plt.figure(figsize=(10,6))
bars = plt.barh(top_urls['url'], top_urls['count'], color="#64cc85")
plt.xlabel("Requests", fontsize=13)
plt.ylabel("URL", fontsize=13)
plt.title("Top 10 Requested URLs", fontsize=15, fontweight='bold')
plt.gca().invert_yaxis()
```

```
for bar in bars:
    plt.text(bar.get_width()+2, bar.get_y()+bar.get_height()/2,
str(int(bar.get_width())), va='center')
plt.tight_layout()
plt.show()

from pyspark.sql.functions import col, coalesce, lit

# Reuse df_with_time which already has the 'hour' column extracted using
PySpark
# Filter for error logs (status >= 400) using PySpark
error_df_spark = df_with_time.filter(col('status') >= 400)

# Calculate total requests per hour using PySpark
total_requests_per_hour_spark =
df_with_time.groupBy('hour').count().orderBy('hour')

# Calculate error requests per hour using PySpark
error_requests_per_hour_spark =
error_df_spark.groupBy('hour').count().orderBy('hour')

# Join the two DataFrames to calculate the error rate
# Use a left outer join from total requests to error requests to include
hours with no errors
error_rate_by_hour_spark = total_requests_per_hour_spark.join(
    error_requests_per_hour_spark,
    on='hour',
    how='left_outer'
).select(
    total_requests_per_hour_spark.hour,
    (coalesce(error_requests_per_hour_spark['count'], lit(0)) /
total_requests_per_hour_spark['count']).alias('error_rate')
).orderBy('hour')

# Convert the result to pandas for plotting
error_by_hour_pandas = error_rate_by_hour_spark.toPandas()

import matplotlib.pyplot as plt

plt.figure(figsize=(10,6))
```

```
plt.plot(error_by_hour_pandas['hour'], error_by_hour_pandas['error_rate'],
marker='o', color='#ffb800', label='Error Rate')
plt.xlabel("Hour of Day", fontsize=13)
plt.ylabel("Error Rate", fontsize=13)
plt.title("Error Rate by Hour", fontsize=15, fontweight='bold')
plt.ylim(0, 1)
plt.xticks(error_by_hour_pandas['hour']) # Ensure all hours with data are
shown on x-axis
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.show()
```




```
top_ips = df.groupBy("ip").count().orderBy("count",
ascending=False).limit(10).toPandas()
plt.figure(figsize=(7,7))
plt.pie(top_ips['count'], labels=top_ips['ip'], autopct='%1.1f%%',
startangle=140, colors=plt.cm.Paired.colors)
plt.title("Top 10 Clients by Number of Requests", fontsize=15,
fontweight='bold')
plt.tight_layout()
plt.show()
```



---

9. Output & Sample Results

(Visualizations—bar charts, line graphs, and pie charts—can be added in this section.)

- **Hourly Request Volume Table**: Shows the distribution of request counts per hour.
- **Status Code Distribution**: Provides a breakdown of response code frequencies (e.g., 200, 404, 500).
- **Top URLs Table**: Lists the most-requested resources.
- **Error Rate Table**: Presents hour-by-hour error percentages, highlighting timeframes of operational concern.
- **Anomaly Alerts**: Examples include alerts like "Error rate exceeded 10% in hour 12" for any window breaching predefined thresholds.

---

10. Conclusion & Future Work

The real-time log analysis pipeline successfully delivers:

- **Scalable, low-latency log ingestion and processing** for modern distributed systems.
- **Near-instant detection** of operational anomalies and error surges.
- **Efficient storage and historical analytics** using HDFS and Hive.
- **Actionable insights** through automatic aggregations and live metrics.

Future Extensions

- Deployment of **advanced ML techniques** (e.g., deep learning or autoencoders) for complex anomaly and threat detection.
- Seamless **cloud integration** (AWS, Azure) for elasticity and managed scaling.
- **NLP-based log field extraction** and semantic classification.
- **Automated remediation actions** on critical alert triggers.

---

11. References

- **Abstract.pdf**: Project summary and breakdown of module design.
- **Algorithm.pdf**: Step-wise algorithm description and pseudocode.
- **Real_time_Log_Analysis_Using_Hadoop_and_Spark.ipynb**: Complete codebase, results, and prototype analytics.
- **Apache Spark, Hadoop, Kafka, Flume Official Documentation**.
- **Grafana, Kibana Documentation** for dashboard and visualization best practices.
- **Industry whitepapers** and **Cloudera/Databricks big data analytics guides**.

Thooti Jayanthi Devi
24M11MC173
Aditya University