



SCHOOL OF SCIENCE & ENGINEERING

FALL 2024

CSC535501 Intro. to Big Data: Environment and Applications

Final Project
Real Time and Batch Access Log Processing

December 2nd, 2024

Realized by:

Noura Ogbi
Salma Kaichouh

Supervised by:

Dr. Tajjeeddine Rachidi

Table of Contents

PROJECT TEAM.....	3
PROJECT DEFINITION	3
PROJECT DESIGN	4
PROJECT IMPLEMENTATION	5
1. Deploying a Cluster of NGNIX Webservers.....	5
2. Deploying a Kafka Broker Cluster.....	5
3. Filebeat Log Shipping Setup	6
4. Deploying Cassandra.....	6
5. Kafka Streams	7
6. Batch Processing	7
7. Testing Everything	8
IMPLEMENTATION LIMITATIONS	9
PROJECT DEMO & CODEBASE	9
REFERENCES	10

<https://github.com/gitNoura/Batch-Stream-log-access>

Refer to README file for more details.

PROJECT TEAM

<i>Task</i>	<i>Team Member</i>
Setup and Configuration	Noura and Salma
Log Streaming and Storage	Noura and Salma
Real-Time Processing	Noura and Salma
Batch Processing	Noura and Salma
Testing and Validation	Noura and Salma
Report	Noura and Salma
Readme File	Noura and Salma

PROJECT DEFINITION

The project aims to build a scalable pipeline for real-time and batch processing of access logs from web server pages, specifically designed to track and report the most visited products/pages. The primary goals and requirements of the project are as follows:

➤ Project Goals

- **Real-Time Processing:** To create a system that can process access logs in real-time, providing insights into the most visited pages every 5 minutes.
- **Batch Processing:** To implement a batch process that aggregates access logs for an entire day, producing a report of the most visited pages/products.
- **Scalability:** To ensure that the pipeline can handle increasing loads and can be easily scaled by adding more web servers or Kafka brokers.
- **Reliability:** To build a robust system that can recover from failures and ensure data integrity.

➤ Functional Requirements

- We need to deploy a cluster of NGINX web servers behind a load balancer to simulate web traffic. The next step is to stream access logs from NGINX to a Kafka broker using a predefined JSON log format. We also need to configure Kafka consumers for:
 - Storing raw logs in a Cassandra NoSQL database.
 - Aggregating and displaying the top N pages accessed within a specified interval.
- Finally, we need to use a batch process to generate daily access reports and store them in Cassandra.

➤ Non-Functional Requirements

- The system should be able to process logs with minimal latency, ensuring real-time insights are delivered promptly. The system also needs to be **Reliable** by implementing fault-tolerant configurations for Kafka and Cassandra to prevent data loss. The architecture should support horizontal scaling, allowing additional web servers and Kafka brokers to be added as needed.

➤ **Architectural Requirements**

- Integration of key components (NGINX, Kafka, Cassandra) to support a seamless pipeline.
- Partitioning and replication in Kafka and Cassandra for performance and fault tolerance.
- Flexible configuration to adjust parameters such as N (number of top pages) and P (time interval for real-time processing).

PROJECT DESIGN

The system architecture employs a distributed data pipeline as shown in the Proposed Solution Architecture. We used Cassandra as the primary NoSQL database and Kafka Streams as the Streaming Engine.

Data Models:

- Access Logs Table: Wide-column structure with timestamp-based partitioning
- Product Stats Table: Counter-based model optimized for aggregation queries
- Schema designed for efficient time-series data storage and retrieval

Data Encoding and Transport:

- JSON format for Nginx logs and Kafka messages
- Binary protocol for Cassandra client communication

Query Patterns:

- Real-time: 5-minute window aggregations for top N products
- Batch: Daily aggregations at 8pm for top N products

PROJECT IMPLEMENTATION

1. Deploying a Cluster of NGINX Webservers

The Nginx deployment consists of a load balancer and multiple backend servers hosting five product endpoints.

The load balancer is configured with worker processes and event handling optimization for high traffic. Backend servers are exact replicas containing product pages 1-5. Access logs are formatted in JSON to streamline integration with Kafka and subsequent processing. Key

```
[q@archlinux ~]$ curl http://localhost:8080/product1
<html>
<head><title>Product 1</title></head>
<body>
  <h1>Welcome to Product 1 Page</h1>
  <p>This is the page for Product 1.</p>
</body>
</html>
[q@archlinux ~]$ curl http://localhost:8080/product5
<html>
<head><title>Product 5</title></head>
<body>
  <h1>Welcome to Product 5 Page</h1>
  <p>This is the page for Product 5.</p>
</body>
</html>
[q@archlinux ~]$ curl http://localhost:8080/product4
<html>
<head><title>Product 4</title></head>
<body>
  <h1>Welcome to Product 4 Page</h1>
  <p>This is the page for Product 4.</p>
</body>
</html>
[q@archlinux ~]$
```

configuration parameters include worker connections set to 1024 and upstream backend servers running on ports 8081 and 8082. To start Nginx, we begin by copying our configuration files to the appropriate locations. For the load balancer configuration, we place the nginx.conf file in /etc/nginx/nginx.conf. For backend servers, we place their respective configuration files in suitable directories. We verify by accessing endpoints:

<http://localhost:8080/product1> for example.

2. Deploying a Kafka Broker Cluster.

The Kafka cluster deployment utilizes three brokers operating on ports 9093 through 9095, ensuring scalability and fault tolerance. Each broker has a unique configuration including distinct broker IDs, log directories, and ports. The RAWLOG topic is configured with three partitions to enable parallel processing of incoming log data.

Apache Zookeeper serves as the centralized coordination service for our Kafka cluster. We started Zookeeper using the default configuration file, which established the service on port 2181. This service maintains the state of the Kafka cluster, tracks node status, and manages configuration updates.

For our Kafka deployment, we configured three brokers running on ports 9093, 9094, and 9095. Each broker received a unique broker.id (1, 2, and 3) and dedicated log directories (/tmp/kafka-logs-1, -2, -3) to prevent storage conflicts. The multi-broker setup enables load distribution and fault tolerance, though with our 8GB RAM constraint, we later optimized to use just two brokers. After establishing the brokers, we created the RAWLOG topic with three partitions to enable parallel processing of incoming log data. The partitioning strategy allows distributed message consumption across multiple consumers in our fan-out configuration, where one consumer writes to Cassandra while another processes real-time aggregations.

3. Filebeat Log Shipping Setup

```
[q@archlinux ~]$ sudo filebeat test output
kafka: localhost:9094...
  parse host... OK
  dns lookup... OK
  addresses: ::1, 127.0.0.1
  dial up... OK
[q@archlinux ~]$ sudo chmod -R 755 /var/log/nginx/
[q@archlinux ~]$ sudo chown -R filebeat:filebeat /etc/filebeat
[q@archlinux ~]$ sudo systemctl restart filebeat
* filebeat.service - Filebeat sends log files to logstash or directly to Elasticsearch.
   Loaded: loaded (/usr/lib/systemd/system/filebeat.service; enabled; preset: disabled)
   Active: active (running) since Sun 2024-11-24 18:30:06 +01; 4s ago
   Invocation: 7e7c8291f64943bbe72af2bb558d24
   Docs: https://www.elastic.co/beats/filebeat
   Main PID: 38880 (filebeat)
   Tasks: 12 (limit: 9183)
   Memory: 35.8M (peak: 36.6M)
   CPU: 227ms
   CGroup: /system.slice/filebeat.service
           └─ssssss /usr/share/filebeat/bin/filebeat -e -c /etc/filebeat/filebeat.yml --path.home /usr/share/filebeat
             --path.config /etc/filebeat --path.data /var/lib/filebeat --path.logs /var/log/filebeat

Nov 24 18:30:09 archlinux filebeat[38880]: {"log.level":"info","@timestamp":"2024-11-24T18:30:09.804+0100","log.logger":"registrar","log.origin":{"file.name":"registrar/registrar.go","file.line":109},"message":"States Loaded from registrar","service.name":"filebeat","ecs.version":"1.6.0"}
Nov 24 18:30:09 archlinux filebeat[38880]: {"log.level":"info","@timestamp":"2024-11-24T18:30:09.804+0100","log.logger":"crawler","log.origin":{"file.name":"beater/crawler.go","file.line":117},"message":"Starting input, keys present on the config: {filebeat.inputs.enabled filebeat.inputs.0.json.add_error_key filebeat.inputs.0.json.keys_under_root filebeat.inputs.0.paths.0 filebeat.inputs.0.type}","service.name":"filebeat","ecs.version":"1.6.0"}
Nov 24 18:30:09 archlinux filebeat[38880]: {"log.level":"warn","@timestamp":"2024-11-24T18:30:09.804+0100","log.logger":"cfgrwarn","log.origin":{"file.name":"log/input.go","file.line":89},"message":"DEPRECATED: Log input. Use Filestream input instead.","service.name":"filebeat","ecs.version":"1.6.0"}
Nov 24 18:30:09 archlinux filebeat[38880]: {"log.level":"info","@timestamp":"2024-11-24T18:30:09.805+0100","log.logger":"input","log.origin":{"file.name":"log/input.go","file.line":171},"message":"Configured paths: [/var/log/nginx/access.log]","service.name":"filebeat","input.id":"d4958acc-444e-4592-bc63-208846a18ff2","ecs.version":"1.6.0"}
Nov 24 18:30:09 archlinux filebeat[38880]: {"log.level":"info","@timestamp":"2024-11-24T18:30:09.805+0100","log.logger":"crawler","log.origin":{"file.name":"beater/crawler.go","file.line":148},"message":"Starting input (ID: 165524309050675851064)","service.name":"filebeat","ecs.version":"1.6.0"}
Nov 24 18:30:09 archlinux filebeat[38880]: {"log.level":"info","@timestamp":"2024-11-24T18:30:09.805+0100","log.logger":"crawler","log.origin":{"file.name":"beater/crawler.go","file.line":106},"message":"Loading and starting inputs completed. Enabled inputs: 1","service.name":"filebeat","ecs.version":"1.6.0"}
Nov 24 18:30:09 archlinux filebeat[38880]: {"log.level":"info","@timestamp":"2024-11-24T18:30:09.805+0100","log.origin":{"file.name":"cfgrfile/reload.go","file.line":164},"message":"Config reloader started","service.name":"filebeat","ecs.version":"1.6.0"}
```

Filebeat serves as the log shipping agent between Nginx and Kafka. The configuration maps Nginx access logs to the RAWLOG Kafka topic. Filebeat monitors the Nginx log directory with appropriate permissions and transforms the logs into a structured format before sending them to Kafka. The setup includes compression and acknowledgment settings to ensure reliable log delivery while optimizing network bandwidth. The image shows output from Filebeat configuration and startup on an Arch Linux system. The logs indicate successful configuration and startup of Filebeat for shipping Nginx logs

to Kafka. The partitioning approach distributes data across nodes using the timestamp as the partition key, ensuring even data distribution and efficient time-based queries.

4. Deploying Cassandra

In the Cassandra deployment section, we created two essential tables within a dedicated keyspace 'logspace'. The tables are called: LOG and RESULTS as shown in the screenshot. The LOG table was designed to store raw access logs with fields including UUID, timestamp, source IP, request details, status code, user agent, and the complete raw message. Initial deployment attempts faced syntax errors in table creation,

```
[q@archlinux ~]$ cqlsh
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.2.0 | Cassandra 5.0.2 | CQL spec 3.4.7 | Native protocol v5]
Use HELP for help.
cqlsh> CREATE KEYSPACE logspace WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
cqlsh> USE logspace;
cqlsh:logspace> CREATE TABLE LOG (
...
... id UUID PRIMARY KEY,
... timestamp TIMESTAMP,
... source_ip TEXT,
... request TEXT,
... status_code INT,
... user_agent TEXT,
... raw_message TEXT
... );
cqlsh:logspace> CREATE TABLE RESULTS (
...
... id UUID PRIMARY KEY,
... timestamp TIMESTAMP,
... batch_or_realtime TEXT,
... source_ip TEXT,
... processed_data TEXT,
... summary JSON
... );
```

particularly with JSON field specifications, which were resolved by adjusting data types. The RESULTS table in our Cassandra implementation was designed to store both real-time and batch processing aggregations with a structure optimized for time-series analytics. After encountering initial syntax errors with JSON data types, we modified the schema to use TEXT for the summary field instead. The table uses a composite primary key with UUID and timestamp, allowing efficient retrieval of aggregated data within specific time windows. Each record includes fields for differentiating between batch and real-time processing results, source IP tracking for analytics, and processed data storage. The table structure supports our requirement to store the N most visited pages for both 5-minute intervals and daily aggregations at 8 PM, with the timestamp field enabling efficient querying for specific time period.

Our Cassandra implementation uses the wide-column store concept, where each row can have different columns, providing flexibility for storing varying log attributes. The LOG table uses a compound primary key (UUID, timestamp) to enable efficient time-series queries while preventing hotspots. Following Cassandra's CAP theorem properties, we chose availability

and partition tolerance over strict consistency, configuring the system with eventual consistency to optimize write performance for high-volume log ingestion.

Next we had to populate the LOG and RESULTS tables. We did it by writing 2 python codes. The first implements a Kafka consumer using Kafka Streams that reads from the RAWLOG topic and writes to Cassandra. Using the kafka-python and cassandra-driver libraries, we established connections to both services. The consumer operates in fan-out mode with a dedicated consumer group 'log_consumer_group', ensuring parallel processing capability. The script deserializes JSON messages from Kafka and transforms them into appropriate data types for Cassandra storage. You can see the RESULTS table filled below.

```
cqlsh> SELECT * FROM logspace.RESULTS;
```

id	batch_or_realtime	processed_data	source_ip	summary	timestamp
3484030d-36fe-4875-a9d9-794153202973	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.844000+0000
01990265-fbb5-48a4-bc49-78f9e90bfdd2	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.855000+0000
5b892b3e-7fab-4113-93d6-23ba6fe31a7e	realtime	Top N Pages Data	unknown	Processed log from source IP unknown	2024-12-06 18:36:14.865000+0000
2b0b0a32-c4f7-4cf0-9ba3-52b59e62bcb4	realtime	Top N Pages Data	unknown	Processed log from source IP unknown	2024-12-06 18:36:14.919000+0000
ccf0b6d3-d163-4a54-909a-4d606f588d1d	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.881000+0000
07db3b51-fe46-4ff8-9f5a-6ca88bd2b967	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.886000+0000
a880fe74-3543-48a6-935a-593b81509b99	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.869000+0000
696eca71-04c3-435a-bbe5-3203795b024f	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.820000+0000
31d186b0-2aa3-4a8d-b38a-38a3d9d63033	realtime	Top N Pages Data	unknown	Processed log from source IP unknown	2024-12-06 18:36:14.905000+0000
90988eac-32d7-41e8-833e-fc244c224a42	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.860000+0000
7d188253-6b40-4e67-b3fa-47cf6beeb982	realtime	Top N Pages Data	127.0.0.1	Sample summary	2024-12-05 20:54:54.953000+0000
05466fd7-7eca-4c50-870c-d246bac48985	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.901000+0000
aeb6e9d5-2301-4e80-b8a7-8a6125987ab3	realtime	Top N Pages Data	unknown	Processed log from source IP unknown	2024-12-06 18:36:14.876000+0000
45244638-de83-4bce-a744-5642b4caa7a4	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.910000+0000
71436748-195d-4d17-9554-44aaba527436	realtime	Top N Pages Data	unknown	Processed log from source IP unknown	2024-12-06 18:36:14.849000+0000
e6033548-686e-4891-86f5-1c1078ae79ee	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.897000+0000
cf9ebd58-6021-496f-9178-0f69c5fb53a1	realtime	Top N Pages Data	127.0.0.1	Sample summary	2024-12-05 20:54:50.920000+0000
aebcf03f-8ccc-4815-a5c3-db4c5cf4e46e	realtime	Top N Pages Data	127.0.0.1	Processed log from source IP 127.0.0.1	2024-12-06 18:36:14.915000+0000
52eb4d6b-54b0-4f80-9ace-2c53ad6b48e4	realtime	Top N Pages Data	unknown	Processed log from source IP unknown	2024-12-06 18:36:14.892000+0000

5. Kafka Streams

The Kafka Streams implementation involved setting up a Java application using Gradle to process incoming log data and calculate real-time page visit statistics. Using Gradle version 8.4, we created a project structure with JUnit Jupiter for testing and Kafka Streams 3.5.0 for stream processing. The build script was configured in Kotlin DSL with Java 11 as the target version.

The application accepts two command-line arguments: N (number of top pages to track) and P (time window in minutes). Key dependencies include kafka-streams for stream processing and junit-jupiter for testing. The project structure follows standard Java application conventions with Maven Central for dependency resolution. In our case, N=3 and P=5, the arguments were given with the “sudo gradle run --args='3 5'”.

The Kafka Streams processor writes aggregated results to the Cassandra RESULTS table through a dedicated Python script. It utilizes a dedicated consumer group 'results_consumer' to read from the RESULTS topic, where Kafka Streams publishes its aggregated page visit statistics. The script deserializes JSON messages and maps them to our Cassandra RESULTS table schema, generating unique UUIDs for each record and timestamping them at insertion. For consistency, it structures both processed data and summary fields as JSON strings before storage. The script maintains continuous operation through an infinite loop, processing each incoming message and executing Cassandra inserts with proper error handling.

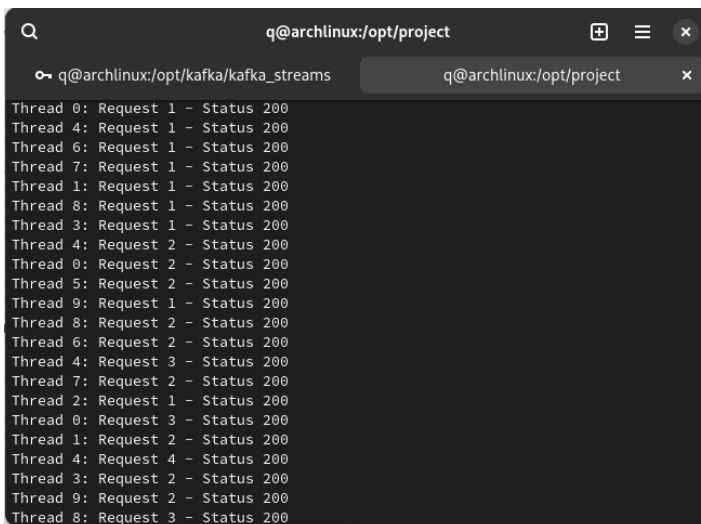
6. Batch Processing

The batch processing component of our pipeline executes daily at 8 PM, analyzing the accumulated log data to identify the most visited pages over the entire day. We implemented a Python script that connects to our Cassandra cluster and processes the LOG table data using a specific time window. The script establishes a connection to Cassandra using the cassandra-

driver library and operates on the 'logspace' keyspace. It processes data with configurable parameters, including N (number of top pages to report) and the current date's log entries. Results are then written back to the LOG table with specific identifiers to distinguish batch processing results from real-time data. Each record includes metadata such as timestamp, source identification (marked as "batch-process"), and visit counts.

We also used a cron implementation for scheduling tasks in Unix-like systems, to automate the execution of our batch processing script. We configured a cron job to run the batch process at 8 PM every day, which processes logs from the LOG table in Cassandra, identifies the most visited pages, and writes the results back into the same table.

7. Testing Everything



```
q@archlinux:/opt/project
q@archlinux:/opt/kafka/kafka_streams
q@archlinux:/opt/project

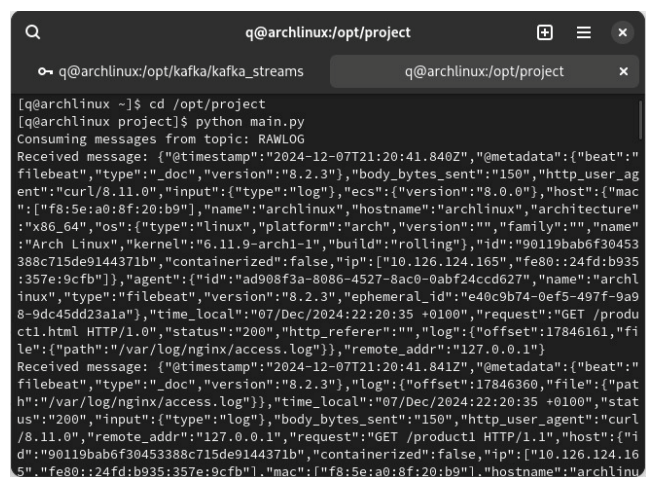
Thread 0: Request 1 - Status 200
Thread 4: Request 1 - Status 200
Thread 6: Request 1 - Status 200
Thread 7: Request 1 - Status 200
Thread 1: Request 1 - Status 200
Thread 8: Request 1 - Status 200
Thread 3: Request 1 - Status 200
Thread 4: Request 2 - Status 200
Thread 0: Request 2 - Status 200
Thread 5: Request 2 - Status 200
Thread 9: Request 1 - Status 200
Thread 8: Request 2 - Status 200
Thread 6: Request 2 - Status 200
Thread 4: Request 3 - Status 200
Thread 7: Request 2 - Status 200
Thread 2: Request 1 - Status 200
Thread 0: Request 3 - Status 200
Thread 1: Request 2 - Status 200
Thread 4: Request 4 - Status 200
Thread 3: Request 2 - Status 200
Thread 9: Request 2 - Status 200
Thread 8: Request 3 - Status 200
```

For this section, we chose to switch to a python code instead of Jmeter for various reasons, including the memory space problems. Thus, we used a python code to send requests that should be received by NGINX that are generated as logs to Kafka. The system then processes those logs through the real-time and batch processing (main.py).

First, we developed a Python script using multithreading to send HTTP requests to an NGINX load balancer,

simulating concurrent users accessing the system (requests_simulator.py). Each thread (10) generated multiple requests (5), and the response status was tracked to verify successful communication with the server. The “200” means that the communication was successful. This simulation ensured that the NGINX load balancer and web servers could handle the traffic as expected. Next, we implemented Python scripts to validate the integration of NGINX, Kafka, and Cassandra. A Kafka consumer script was used to confirm that logs generated by NGINX were successfully forwarded to the Kafka RAWLOG topic via Filebeat (kafka_consumer.py). The consumer displayed the 5 first messages from the topic to verify the log ingestion. Additionally, we used another Python script to monitor real-time processing results from Kafka's PRODUCTS topic or query the Cassandra RESULTS table for both real-time and batch processing outputs. These scripts allowed us to confirm the functionality of the end-to-end pipeline, ensuring that logs were processed correctly and that system performance metrics, such as throughput and request handling, met the project's requirements.

Each component of our pipeline underwent specific unit testing to ensure proper functionality before integration. For the Nginx setup, we



```
q@archlinux:/opt/project
q@archlinux:/opt/kafka/kafka_streams
q@archlinux:/opt/project

[q@archlinux ~]$ cd /opt/project
[q@archlinux project]$ python main.py
Consuming messages from topic: RAWLOG
Received message: {"@timestamp":"2024-12-07T21:20:41.840Z","@metadata":{"beat":"filebeat","type":"doc","version":"8.2.3"},"body_bytes_sent":150,"http_user_agent":"curl/8.11.0","input":{"type":"log"},"ecs":{"version":"8.0.0"},"host":{"mac":"f8:5e:a0:8f:20:b9"},"name":"archlinux","hostname":"archlinux","architecture":"x86_64","os":{"type":"linux","platform":"arch","version":"","family":"","name":"Arch Linux","kernel":"6.11.9-arch1-1","build":"rolling"},"id":"90119bab6f30453388c715de9144371b","containerized":false,"ip":["10.126.124.165"],"fe80::24fd:b935:357e:9c9b"},"agent":{"id":"ad908f3a-8086-4527-8ac0-0abf24ccd627","name":"archlinux","type":"filebeat","version":"8.2.3","ephemeral_id":"e40c9b74-0ef5-497f-9a98-9dc45dd23a1a"},"time_local":"07/Dec/2024:22:20:35 +0100","request":"GET /product1.html HTTP/1.0","status":"200","http_referer":"","log":{"offset":"17846161","file":{"path":"/var/log/nginx/access.log"},"remote_addr":"127.0.0.1"}
Received message: {"@timestamp":"2024-12-07T21:20:41.841Z","@metadata":{"beat":"filebeat","type":"doc","version":"8.2.3"},"log":{"offset":"17846360","file":{"path":"/var/log/nginx/access.log"},"time_local":"07/Dec/2024:22:20:35 +0100"},"status":"200","input":{"type":"log"},"body_bytes_sent":150,"http_user_agent":"curl/8.11.0","remote_addr":"127.0.0.1","request":"GET /product1 HTTP/1.1","host":{"id":"90119bab6f30453388c715de9144371b","containerized":false,"ip":["10.126.124.165"],"fe80::24fd:b935:357e:9c9b"},"mac":["f8:5e:a0:8f:20:b9"],"hostname":"archlinux"}
```


tested the load balancer configuration and verified individual server responses through curl commands to each product endpoint.

The Kafka cluster testing involved verifying broker connectivity and topic creation using kafka-topics.sh scripts. We tested consumer functionality in isolation, confirming message delivery and retention across partitions. Each broker was individually tested for proper startup and shutdown sequences, with logs analyzed for any configuration issues.

For Cassandra, we developed test cases using the CQL shell to verify table schemas and data insertion patterns. Test queries included checking primary key constraints, verifying timestamp-based queries, and validating JSON data storage in both LOG and RESULTS tables. Sample data insertion and retrieval tests confirmed proper data type handling and index performance.

The Filebeat configuration underwent testing through log file monitoring and message delivery verification. We created test log entries in various formats to ensure proper parsing and JSON transformation. Error handling was verified by introducing malformed log entries and monitoring Filebeat's response.

All test results were documented in the README file, please refer to it for more details.

IMPLEMENTATION LIMITATIONS

During the implementation of our pipeline, we encountered a lot of technical challenges and limitations. One of the constraints was our system's 8GB RAM limitation, which created substantial performance bottlenecks. Running the complete stack - Nginx, Kafka cluster, Cassandra, and JMeter - simultaneously led to shutdowns. This forced us to optimize our deployment strategy, reducing the number of Kafka brokers and using a Python code instead of Jmeter.

Version compatibility presented another major challenge across multiple components. We faced Python version conflicts with Kafka, requiring us to create a dedicated virtual environment to manage dependencies effectively. The 'six.moves' library in Kafka caused many issues until we modified the main Kafka Python configuration file by adding the library there. Additionally, Cassandra compatibility required downgrading our Java version and operating with pyenv global 3.11.6.

One persistent issue we couldn't resolve involved Filebeat's JSON decoding capabilities. The error "Error decoding JSON: json: cannot unmarshal number into Go value of type map[string]interface { }" occurred when Filebeat attempted to process certain log entries. This limitation affected our log processing pipeline's reliability, particularly when handling numeric values in log messages. Despite multiple attempts at configuration adjustments, this remained an unresolved challenge in our implementation.

PROJECT DEMO & CODEBASE

Video Link: <https://www.youtube.com/watch?v=jclXQ4VIZ6U&feature=youtu.be>

Codebase can be found in the .zip file.

REFERENCES

- [1] Kleppmann, Martin. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017, Chapter 10.
- [2] Kleppmann, Martin. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017, Chapter 11.
- [3] Carpenter, J., & Hewitt, E. (2022). *Cassandra: The definitive guide* (3rd ed.). O'Reilly Media.
- [4] Palino, T. (2022). *Stream processing with Apache Kafka: Building real-time data systems by example*. O'Reilly Media.
- [5] Apache Cassandra Documentation. (2024). <https://cassandra.apache.org/doc/latest/>
- [6] NGINX Documentation. (2024). <https://nginx.org/en/docs/>
- [7] Elastic Documentation - Filebeat Reference. (2024). <https://www.elastic.co/guide/en/beats/filebeat/current/>