Log Streaming Pipeline Documentation

Table Content

1. Problem Statement

The purpose of this exercise is for you to demonstrate your technical abilities in setting up an end-to-end data pipeline. Some of the technologies we'd like you to use are specific, while others we leave open for interpretation, so you will get to choose.

From a high level, we will provide you with a dataset that you will produce to a message queue, consume that dataset from the message queue do some manipulations, and send it downstream to a datastore. This should all be done locally via a set of docker containers that communicate with each other.

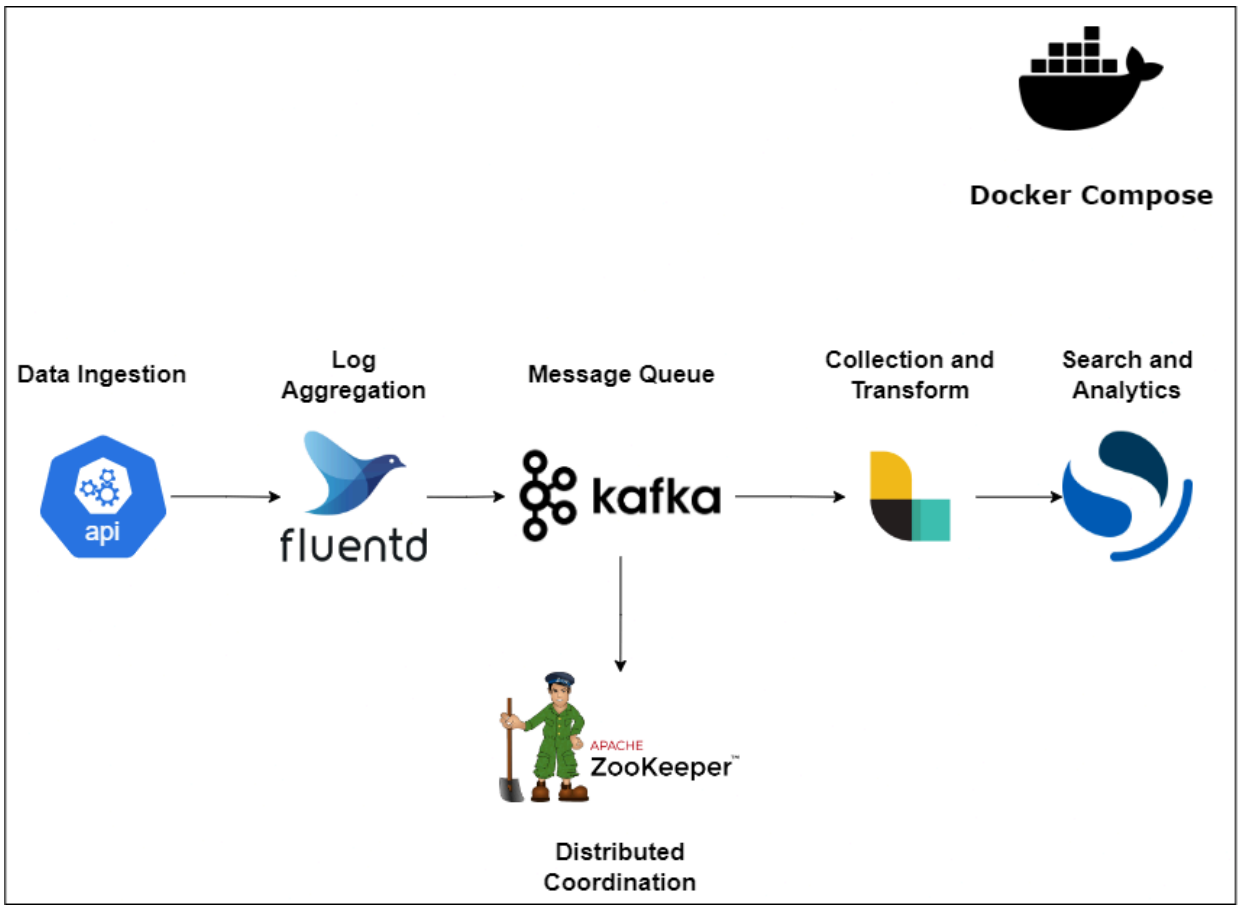
The deliverables for this assessment are a GitHub repository with the following:
1. A docker compose file that will stand up your data pipeline end-to-end.
2. A set of helper scripts that will help you interact with the data pipeline.
Potential helper scripts could do the following:
    a. Start the data pipeline.
    b. Stop the data pipeline.
    c. Produce the events to the data pipeline.
    d. Monitor the data pipeline.
    e. Give the status of all of the components of the data pipeline.

3. A README that will document how to use the docker-compose file and helper scripts.
4. A design document describing your technical solution and an architecture diagram.

2. Solution

The log streaming pipeline is designed to collect, process, store, and visualize logs generated by a web application. It utilizes various components such as Fluentd, Kafka, Logstash, OpenSearch, Loki, Promtail, and Grafana to achieve these tasks.

3. High-Level Architecture

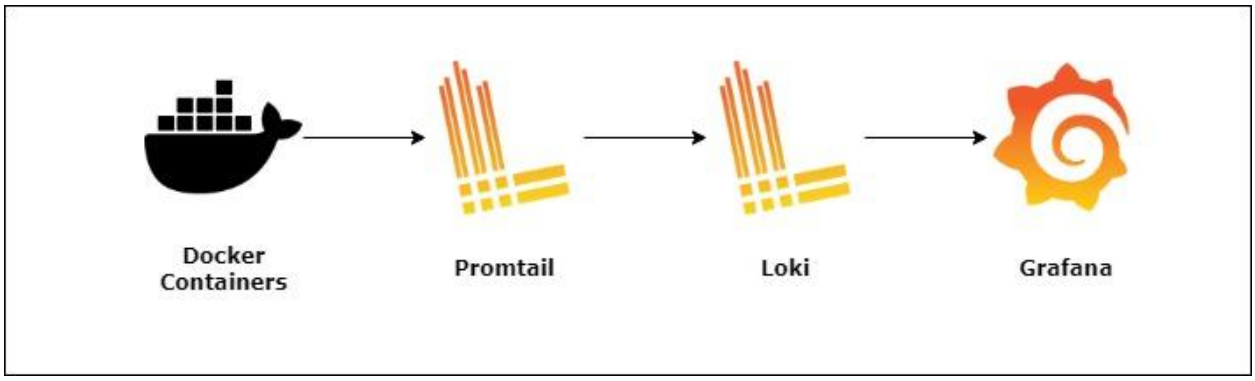


Below is a high-level overview of the architecture:

1. **Web application**: Created by using FastApi which helps to generate logs to be collected and processed.
2. **Fluentd**: Collects logs from the web application and forwards them to Kafka for processing.
3. **Kafka**: Acts as a message broker for log messages, providing a scalable and fault-tolerant platform for log processing.
4. **Zookeeper**: Coordination service for Kafka

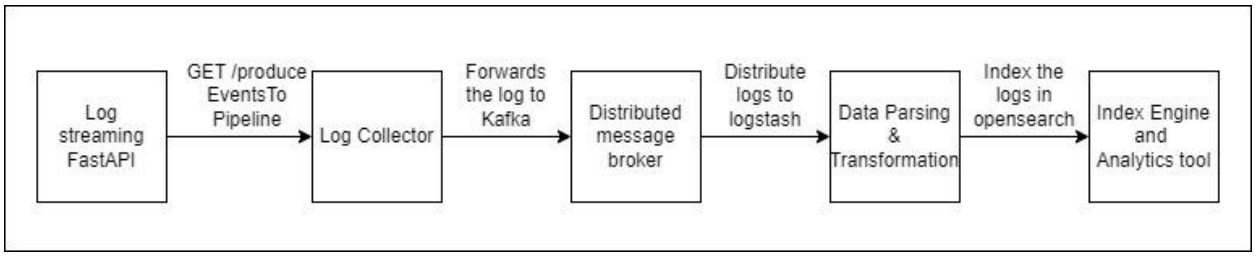
5. **Logstash**: Processes logs received from Kafka, allowing for filtering, parsing, and transforming log data before sending it to OpenSearch for storage.
6. **OpenSearch**: Stores processed logs and enables efficient searching and analysis of log data.

Additional Docker Container Log Pipeline:



Promtail: Ships logs from the containers to Loki for aggregation.
Loki: Provides log aggregation capabilities, allowing for efficient storage and querying of logs.
Grafana: Provides a visualization dashboard for analyzing log data stored in Loki.

4. Low-Level Architecture



5. Components Explanation and Configuration

01. Web Application:

1. Logs Setup in ‘main.py’:

- The web application, built using FastAPI, includes logging functionality to stream logs to Fluentd.
- Logs are formatted in JSON and fetched from a sample NGINX log file hosted on GitHub.

2. Endpoint '/produceEventsToPipeline':
- An endpoint '/produceEventsToPipeline' is defined to initiate log streaming to Fluentd.
- Logs are fetched asynchronously using 'httpx.AsyncClient' from the NGINX log file.
- Each log entry is parsed as JSON and sent to Fluentd using the 'fluent.sender' module.

3. Fluentd Configuration:
- The Fluentd sender is configured to communicate with the Fluentd instance using the host 'fluent' and port '24224'.

4. Error Handling:
- Exception handling is implemented to manage errors during log fetching and streaming.
- If an error occurs, it is logged, and an appropriate error message is returned to the client.

02. Fluentd:

1. Input Source Configuration:
- Fluentd is configured to listen for incoming logs on port 24224 from all network interfaces (bind "0.0.0.0").

2. Output Configuration for 'fluent.info' Logs:
- Logs tagged as 'fluent.info' are directed to the standard output ('@type stdout'), displaying them in the console.

3. Output Configuration for Application Logs ('app.**'):
- Logs from the web application are sent to Kafka brokers ('@type kafka_buffered') located at 'kafka-node1:9092'.
- Buffered output ('<buffer>') is used with file-based buffering, storing logs at '/fluentd/buffer/kafka'.

- ‘flush_thread_count’: Two threads are utilized for flushing buffered data.
- ‘flush_interval’: Buffered data is flushed to Kafka every 5 seconds (`5s`).

4. Key Configurations:
- ‘@type forward’ for input from remote sources.
- ‘@type kafka_buffered’ for sending logs to Kafka with buffering.
- ‘port 24224’ for the Fluentd input port.
- ‘Brokers Kafka-node1:9092’ specifies the Kafka broker address.
- The ‘<buffer>’ section defines buffering options for Kafka output.

This configuration ensures efficient log collection from web application, and reliable transmission to both the console and Kafka for further processing and analysis.

03. Kafka:

1. Broker Configuration:
- Kafka broker ID (‘KAFKA_BROKER_ID’) is set to 1.
- Advertised listener configuration (‘KAFKA_ADVERTISED_LISTENERS’) specifies the hostname and port for clients to connect (‘PLAINTEXT://kafka-node1:9092’).
- Listener security protocol map (‘KAFKA_LISTENER_SECURITY_PROTOCOL_MAP’) defines the security protocol for communication (‘PLAINTEXT:PLAINTEXT’).
- Advertised hostname (‘KAFKA_ADVERTISED_HOST_NAME’) is set to `kafka-node1`.
- Zookeeper connection string (‘KAFKA_ZOOKEEPER_CONNECT’) points to the Zookeeper service (‘zookeeper-node1:2181’).

2. Topic Configuration:
- Kafka is configured to create a topic named ‘nginx-logs’ with a single partition and replication factor (‘KAFKA_CREATE_TOPICS: "nginx-logs:1:1"’).
- Offset topic replication factor (‘KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR’) is set to `1`.

3. Dependencies and Networking:
- Kafka service depends on Zookeeper for coordination (‘depends_on: - zookeeper-node1’).
- It is connected to the custom network ‘my_network’.

This configuration ensures the proper setup and configuration of the Kafka service within the log streaming pipeline, enabling efficient message brokering and communication between components.

04. Logstash:

Logstash is used for consuming the logs from Kafka and indexing it in OpenSearch. Few transformations are done to convert it into the desired index.

Transformation Explanation:

1. Input Configuration:
  - The ‘Kafka’ input plugin is configured to consume messages from the Kafka topic ‘nginx-logs’.
  - Kafka connection details such as bootstrap servers, client ID, group ID, and consumer threads are specified.

2. Filtering with Ruby:
  - A ‘ruby’ filter is applied to perform custom transformations on each event.
  - The Ruby code generates random values for fields ‘region’ and ‘assetid’.
  - The ‘@timestamp’ field is converted to milliseconds and stored in a new field ‘time’.
  - The event structure is modified to include specific fields (‘remote_ip’, ‘remote_user’, ‘request’, ‘response’, ‘bytes’, ‘referrer’, ‘agent’) within an ‘event’ object.

3. Mutating the Event:
  - A ‘mutate’ filter is used to add metadata fields ‘sourcetype’ and ‘index’.
  - Fields related to individual log attributes are removed from the event's outer structure, leaving only the ‘event’ object.

4. Output Configuration:
  - Transformed events are sent to OpenSearch for indexing.
  - The OpenSearch index ‘nginx-logs’ is specified as the destination.

This Logstash configuration demonstrates comprehensive data transformation and enrichment before indexing logs into OpenSearch, ensuring structured and meaningful log data for efficient search and analysis.

Example of one JSON object stored in ‘nginx-logs’ index:


```
{
  "_index": "nginx-logs",
  "_id": "f3OUCI8BOMi1CzXkfa1S",
  "_version": 1,
  "_score": 0,
  "_source": {
    "event": {
      "bytes": 3316,
      "referrer": "-",
      "remote_ip": "217.168.17.5",
      "response": 200,
      "remote_user": "-",
      "agent": "-",
      "request": "GET /downloads/product_2 HTTP/1.1"
    },
    "fields": {
      "region": "us-west-1",
      "assetid": "554878db4d92f4598e34fcf813a98c9c"
    },
    "@version": "1",
    "@timestamp": "2024-04-23T01:31:39.995470770Z",
    "time": 1713835899995,
    "index": "nginx",
    "sourcetype": "nginx"
  },
  "fields": {
    "@timestamp": [
      "2024-04-23T01:31:39.995Z"
    ]
  }
}
```


05. Opensearch:

1. OpenSearch Node (‘opensearch-node1’):
   - The OpenSearch Docker container is based on the ‘opensearchproject/opensearch:2.9.0’ image.
   - It is configured with the necessary environment variables, including cluster and node names, and memory settings.

- Volume mounting is utilized to persist data and logs.
- The container is exposed on ports ‘9200’ and ‘9600’

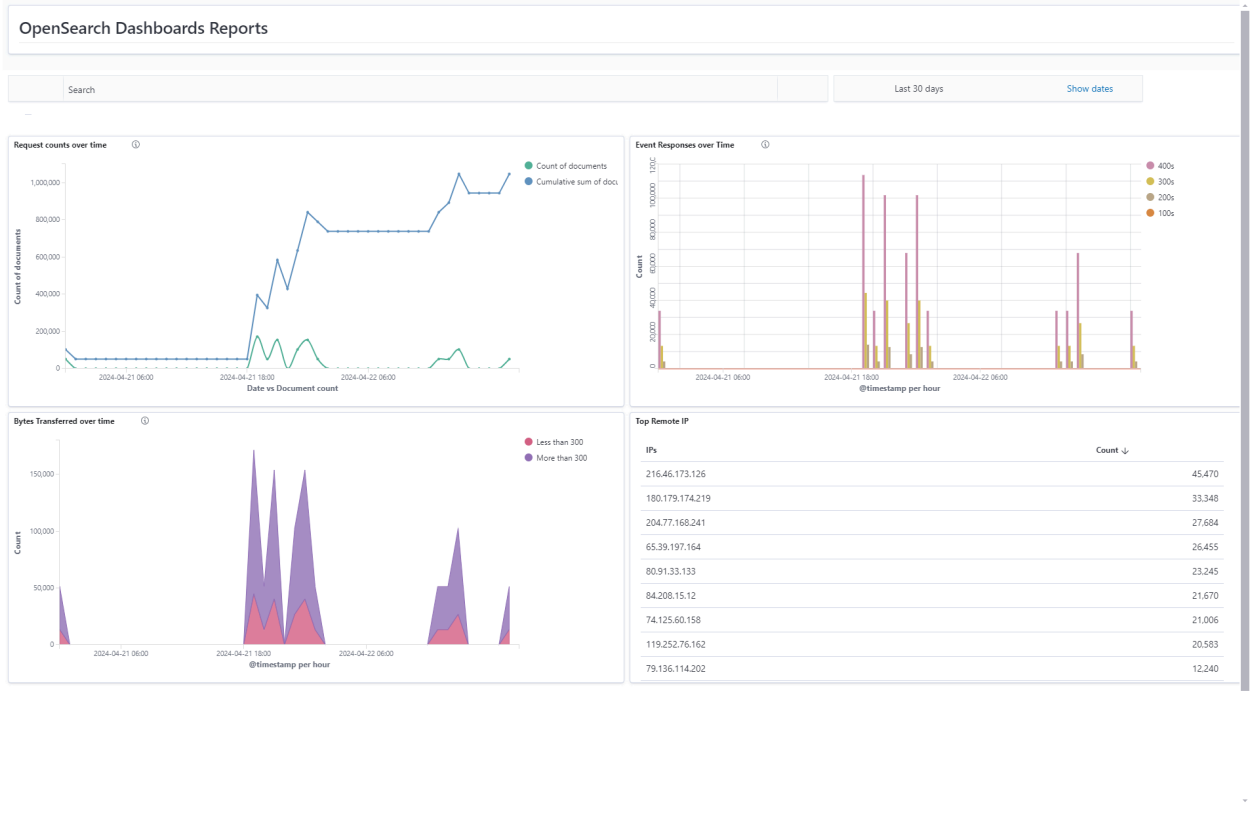
2. OpenSearch Dashboards:
- The OpenSearch Dashboards Docker container is based on the ‘opensearchproject/opensearch-dashboards:2.9.0’ image.
- It is configured to run on port ‘5601’ for accessing the visualization dashboard.

These configurations ensure the deployment of OpenSearch and OpenSearch Dashboards, providing a scalable and user-friendly interface for data visualization and analysis within the log streaming pipeline.

6. Final Product

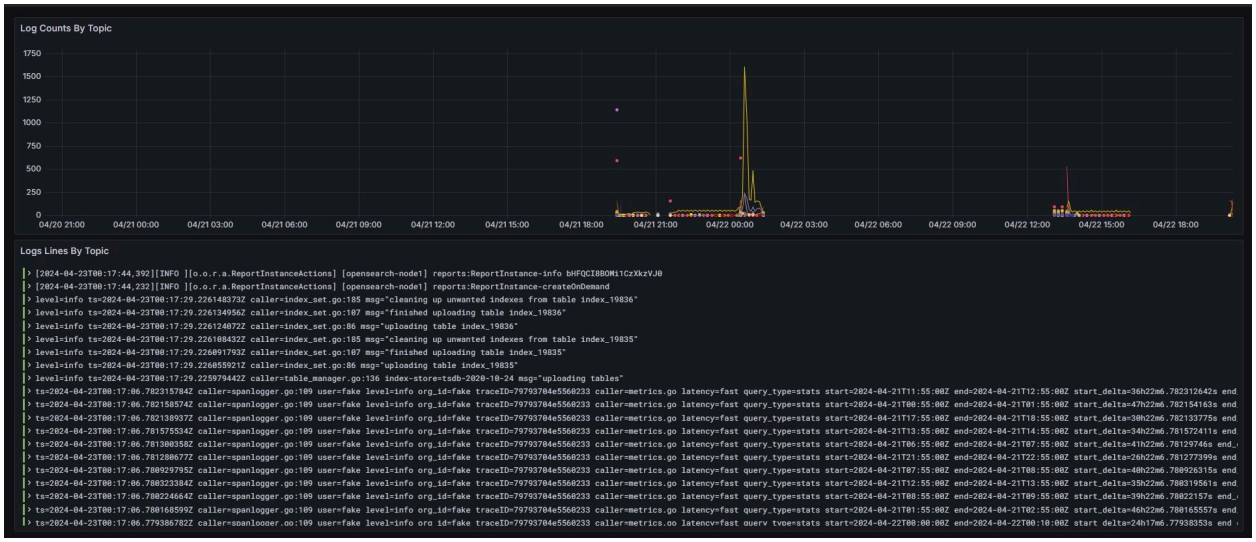
1. Opensearch Dashboard:

This dashboard gives insights on nginx logs data



2. Grafana Dashboard:

This dashboard gives insights into the docker container logs of pipeline components.


Log Counts By Topic
1750
1500
1250
1000
750
500
250
0
04/20 21:00
04/21 00:00
04/21 03:00
04/21 06:00
04/21 09:00
04/21 12:00
04/21 15:00
04/21 18:00
04/21 21:00
04/22 00:00
04/22 03:00
04/22 06:00
04/22 09:00
04/22 12:00
04/22 15:00
04/22 18:00
Logs Lines By Topic
[2024-04-23T00:17:44,392][INFO][o.o.r.a.ReportInstanceActions] [opensearch-node1] reports:ReportInstance-info bHFQCI8BOMi1CzXkzVJ0
[2024-04-23T00:17:44,232][INFO][o.o.r.a.ReportInstanceActions] [opensearch-node1] reports:ReportInstance-createOnDemand
level=info ts=2024-04-23T00:17:29.226148373Z caller=index_set.go:185 msg="cleaning up unwanted indexes from table index_19836"
level=info ts=2024-04-23T00:17:29.226134956Z caller=index_set.go:107 msg="finished uploading table index_19836"
level=info ts=2024-04-23T00:17:29.226124072Z caller=index_set.go:86 msg="uploading table index_19836"
level=info ts=2024-04-23T00:17:29.226108432Z caller=index_set.go:185 msg="cleaning up unwanted indexes from table index_19835"
level=info ts=2024-04-23T00:17:29.226091793Z caller=index_set.go:107 msg="finished uploading table index_19835"
level=info ts=2024-04-23T00:17:29.226055921Z caller=index_set.go:86 msg="uploading table index_19835"
level=info ts=2024-04-23T00:17:29.225979442Z caller=table_manager.go:136 index-store=tsdb-2020-10-24 msg="uploading tables"
ts=2024-04-23T00:17:06.782315784Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T11:55:00Z end=2024-04-21T12:55:00Z start_delta=36h22m6.782312642s end
ts=2024-04-23T00:17:06.782158574Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T00:55:00Z end=2024-04-21T01:55:00Z start_delta=47h22m6.782154163s end
ts=2024-04-23T00:17:06.782138937Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T17:55:00Z end=2024-04-21T18:55:00Z start_delta=30h22m6.782133775s end
ts=2024-04-23T00:17:06.781575534Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T13:55:00Z end=2024-04-21T14:55:00Z start_delta=34h22m6.781572411s end
ts=2024-04-23T00:17:06.781300358Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T06:55:00Z end=2024-04-21T07:55:00Z start_delta=41h22m6.78129746s end_
ts=2024-04-23T00:17:06.781280677Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T21:55:00Z end=2024-04-21T22:55:00Z start_delta=26h22m6.781277399s end
ts=2024-04-23T00:17:06.780929795Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T07:55:00Z end=2024-04-21T08:55:00Z start_delta=40h22m6.780926315s end
ts=2024-04-23T00:17:06.780323384Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T12:55:00Z end=2024-04-21T13:55:00Z start_delta=35h22m6.780319561s end
ts=2024-04-23T00:17:06.780224664Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T08:55:00Z end=2024-04-21T09:55:00Z start_delta=39h22m6.78022157s end_
ts=2024-04-23T00:17:06.780168599Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-21T01:55:00Z end=2024-04-21T02:55:00Z start_delta=46h22m6.780165557s end
ts=2024-04-23T00:17:06.779386782Z caller=spanlogger.go:109 user=fake level=info org_id=fake traceID=79793704e5560233 caller=metrics.go latency=fast query_type=stats start=2024-04-22T00:00:00Z end=2024-04-22T00:10:00Z start_delta=24h17m6.7793833s end