HPE-CTY PROJECT REPORT

Team 04

Project Title:

Real-Time Streaming of Events from Kafka to Elasticsearch Using Kafka Connect

Objective :

To design and implement a real-time data streaming pipeline that ingests events from open sources like Wikipedia, processes them using Kafka, and sinks them into Elasticsearch using Kafka Connect. The system is containerized with Docker and Kubernetes, emphasizing scalability, fault-tolerance, and modular development.

Team Members:

Amith M S Gowda	4VV22CS013
Amrutha R	4VV22CS015
Nagapriya N	4VV22IS061
S Vinod Raj	4VV22CS128
Sumukha S	4VV22CI110

HPE Mentor:

Mr. Hareesh Joshi

hareesh.joshi@hpe.com

College Mentor:

Dr. Vidyashree K P

Associate Professor, Dept of ISE, VVCE

Contents

Supported Features

- KIND-based Kubernetes cluster setup
- Kafka, Elasticsearch, Prometheus, and Grafana deployment
- Real-time data ingestion and processing
- Custom Python services with Prometheus metrics
- Exporter-based monitoring (Kafka, Elasticsearch)
- Grafana dashboards for live system metrics

Technical Approach (Tech Stack)

- **Python:** For data producer (`producer.py`) and consumer (`consumer.py`)
- **Docker:** For containerization
- **Kubernetes (KIND):** For orchestration
- **Apache Kafka:** For distributed message ingestion
- **Elasticsearch:** As the searchable data store
- **Grafana:** For dashboards and alerts
- **Prometheus:** For system monitoring

1 Prototype Phase: Twitter Streaming Integration

Before finalizing Wikipedia as the primary data source, we initially implemented a Twitter-based real-time streaming prototype using the Twitter Developer Platform (formerly Twitter API, now X Developer Platform).

1.1 Setup and Working

We created a Developer Account on X (Twitter) and registered an application to obtain API keys and tokens.

Using Tweepy (a Python library), we authenticated with the API and streamed tweets in real-time based on filters like hashtags and keywords.

Tweets were published to a Kafka topic named `twitter_stream`. Using Kafka Connect, these tweets were automatically pushed to Elasticsearch for indexing and searching.

We visualized the tweet data using Kibana, focusing on tweet text, author, hashtags, and timestamp.

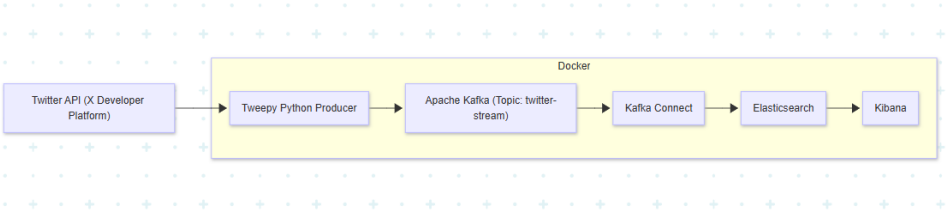


Figure 1: Prototype architecture for Twitter streaming integration.

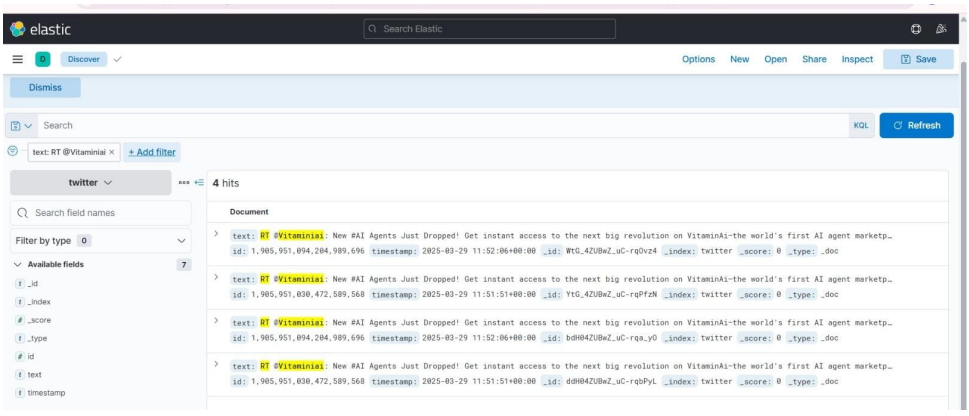


Figure 2: Visualizations of Twitter streaming-events.

1.2 Challenges and Limitations

- **API Rate Limits**: The free-tier Twitter API allowed only limited access (number of tweets per window), which affected continuous streaming.

- **Authentication Overhead**: Required access tokens and secret keys, which made deployment slightly more complex.
- **Connection Stability**: Streaming connection using Tweepy would break occasionally, requiring reconnection logic.

1.3 Why We Switched to Wikipedia

- Wikipedia's EventStreams API is public, open, and doesn't require authentication.
- Offers consistent, continuous real-time data with no quota limits.
- Eliminated the need to manage tokens or rate handling.

2 Prerequisite Setup

2.1 Docker Installation

Install Docker Engine and Docker CLI to build and run container images for Kafka, Zookeeper, Elasticsearch, Grafana, Prometheus, and custom Python applications.

For Docker installation, refer to the official Docker Installation Guide.

Figure 3: Representation of initial project phase using docker kubernetes.

2.2 Kubernetes Environment Setup

Choose a local Kubernetes cluster solution. The project was tested on three options:

a) **Docker Desktop Kubernetes (Initial Setup)**

- Enabled Kubernetes from Docker Desktop settings
- Suitable for quick prototyping

b) **Minikube**

- Installed via package manager or binary
- Allows better resource control and local image builds

c) **KIND (Kubernetes IN Docker) — Recommended**

- Installed using Chocolatey or precompiled binary
- Used to create multi-node clusters using Docker containers

2.3 Kubernetes CLI (kubectl)

Install the `kubectl` command-line tool to manage Kubernetes clusters.

For kubectl installation, refer to the official kubectl-cli Installation Guide.

2.4 Setup Python Environment, Makefile

Install Python 3.7+ and required libraries:

```
pip install kafka-python elasticsearch prometheus_client
```

Custom scripts:

- `wiki_producer.py`
- `wiki_stream_processor.py`

Makefile is used to automate the compilation and linking process, saving from manually typing lengthy commands while execution.

3 Kubernetes Deployment Strategy

3.1 Phase 1: Docker Desktop Kubernetes

Development began with the Kubernetes distribution built into Docker Desktop, offering a simple, single-node environment ideal for rapid prototyping and ease of use.

Benefits:

- Rapid Prototyping: Quick image builds and deployments enabled fast iteration.
- Local Debugging: Easy access to services via `kubectl`, port-forwarding, and NodePorts.
- Familiar Interface: Docker Desktop provided a user-friendly GUI for cluster management.

Limitations:

- High Resource Usage: Significant CPU/RAM consumption led to sluggish performance.
- Unreliable Storage: Persistent Volumes were unstable, risking data loss after restarts.
- Networking Issues: DNS resolution problems affected service discovery and monitoring.
- Single-Node Constraints: Inability to simulate multi-node behavior or test advanced scheduling.

3.2 Phase 2: Migration to Minikube

To overcome Docker Desktop Kubernetes limitations—especially persistent volume instability—the project migrated to Minikube, offering greater flexibility and resource control.

Key Improvements with Minikube:

- Resource Configuration: Explicit CPU and memory allocation improved cluster performance.
- In-Cluster Image Builds: Enabled using `eval $`$minikube docker - env$, simplifying build-deploy cycles.
- Driver Flexibility: Support for various drivers (VirtualBox, KVM, etc.) improved OS compatibility.

Remaining Challenges:

Persistent volume issues persisted for stateful apps like Elasticsearch due to hostPath storage limitations, causing occasional data loss or mount failures after restarts.

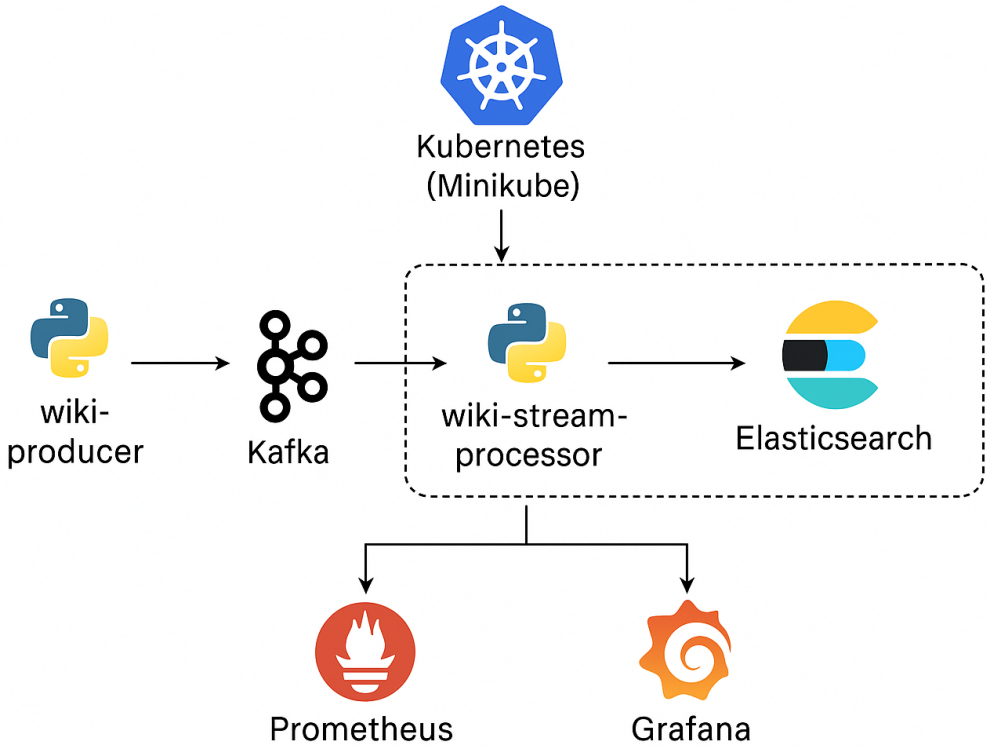


Figure 4: Technical Architecture Overview-Minikube

3.3 Phase 3: Final shift to kind(Kubernetes in docker) service

To resolve persistent volume issues and optimize for CI/CD, the project transitioned to KIND, which runs Kubernetes clusters inside Docker containers.

Why this service was chosen:

- Stable Persistent Volumes: KIND provided more reliable volume mounting for stateful apps like Elasticsearch and Kafka, reducing data loss across restarts.
- Seamless Docker Integration: Local images were easily loaded into the cluster without needing an external registry.
- Lightweight Footprint: KIND used fewer resources than Minikube or Docker Desktop, making it ideal for local development.
- Multi-Node Support: Simple config files enabled multi-node clusters, allowing distributed system testing.

- CI/CD Friendly: KIND worked well in containerized CI environments, enabling consistent and fast automated testing.

This transition enhanced development reliability and aligned the platform with production-like and automated testing workflows.

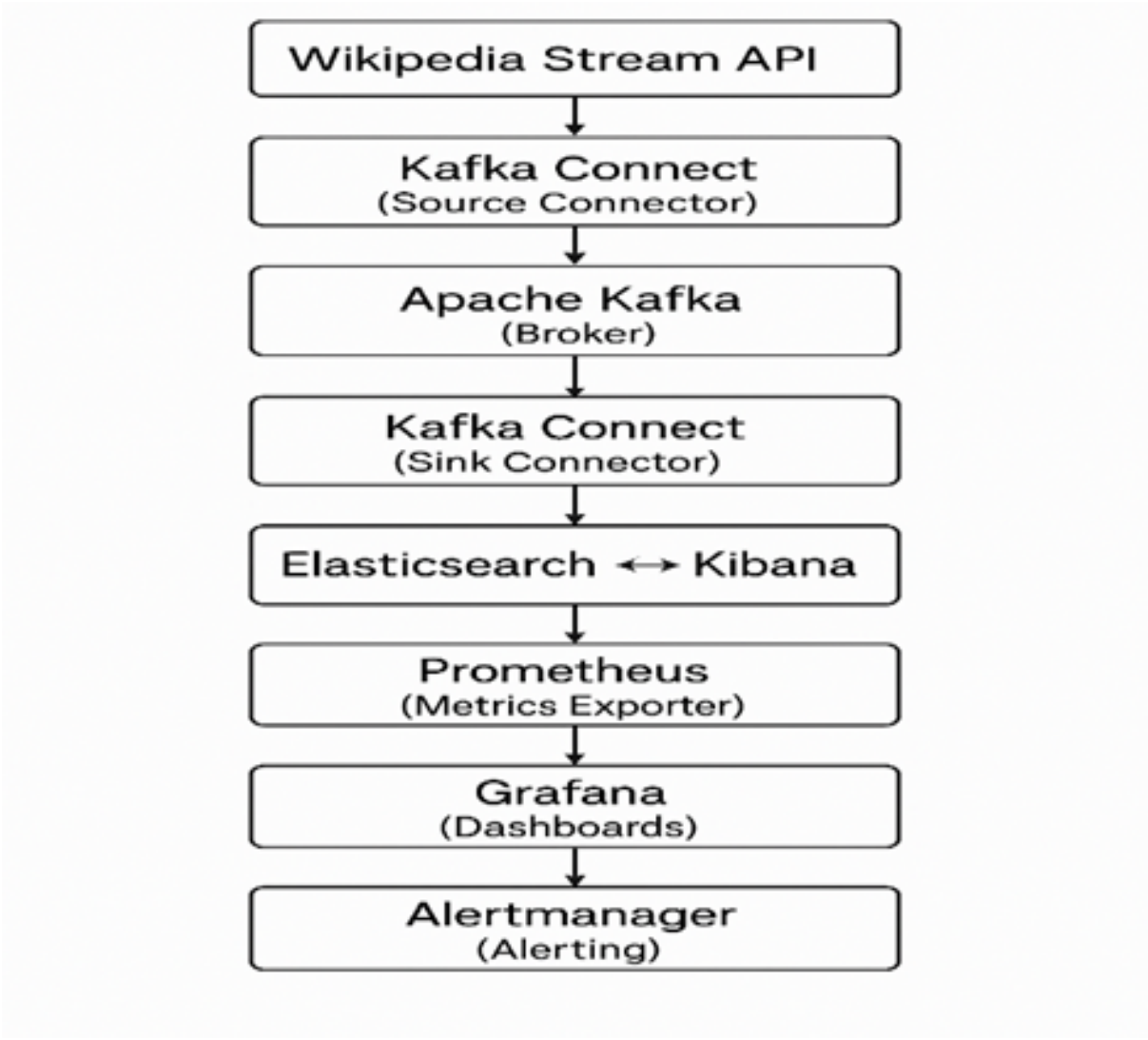


Figure 5: Overview of the System Data Flow

Key Components of the Architecture:

- **Kafka:** A distributed event streaming platform used for high-throughput, real-time data ingestion and processing.
- **Kafka Connect:** A tool for scalable and fault-tolerant data movement between Kafka and external systems like databases or Elasticsearch.
- **Elasticsearch:** A powerful search and analytics engine used for indexing, querying, and visualizing streaming data.
- **Prometheus:** An open-source monitoring system that collects, stores, and queries time-series metrics data.
- **Grafana:** A visualization tool that builds interactive dashboards from data sources like Prometheus.
- **Kafka Topic - wikiTopic:** A Kafka topic used to publish real-time Wikipedia edit events streamed by the producer.

4 Environment Setup

4.1 Multi-node Cluster Configuration

To simulate a realistic distributed environment, a multi-node KIND cluster was created.

The cluster was initialized using:

```
kind create cluster --config kind-config.yaml
```

Later with the help of Makefile Script, made the execution easier.

Make command for creating cluster:

```
make create-cluster
```

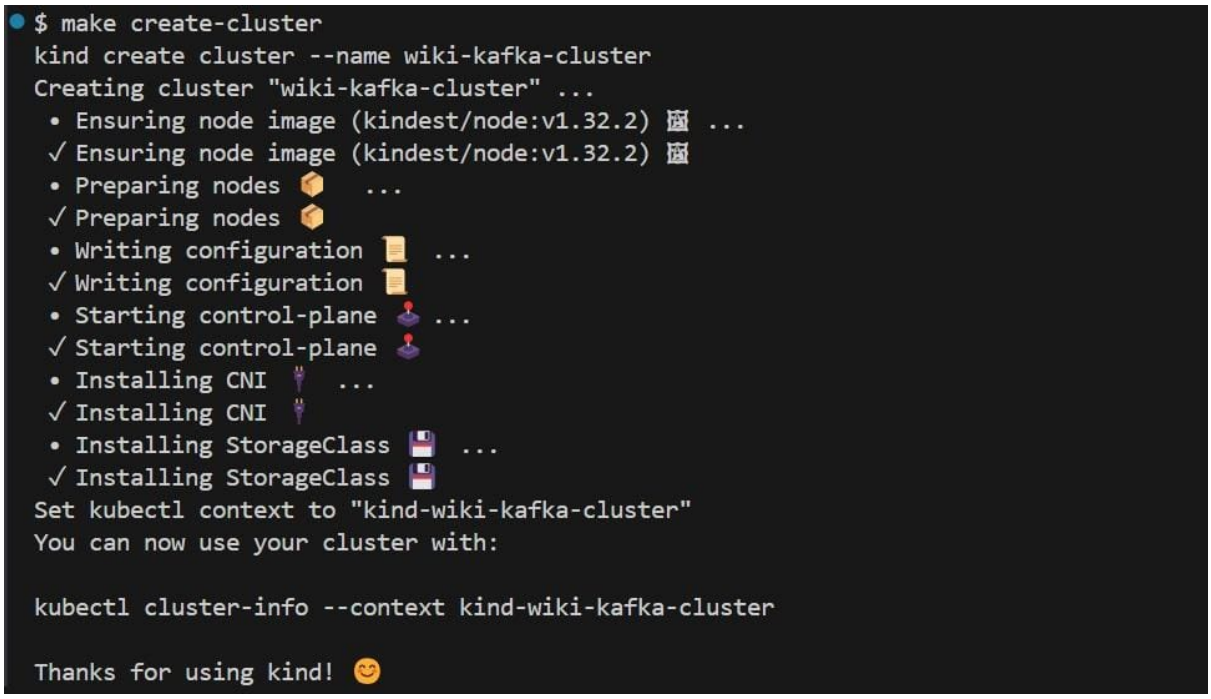


Figure 6: Cluster Creation

4.2 Service Image Loading

Custom applications (`wiki-producer` and `wiki-stream-processor`) were built locally and pushed directly into the cluster without needing an external registry:

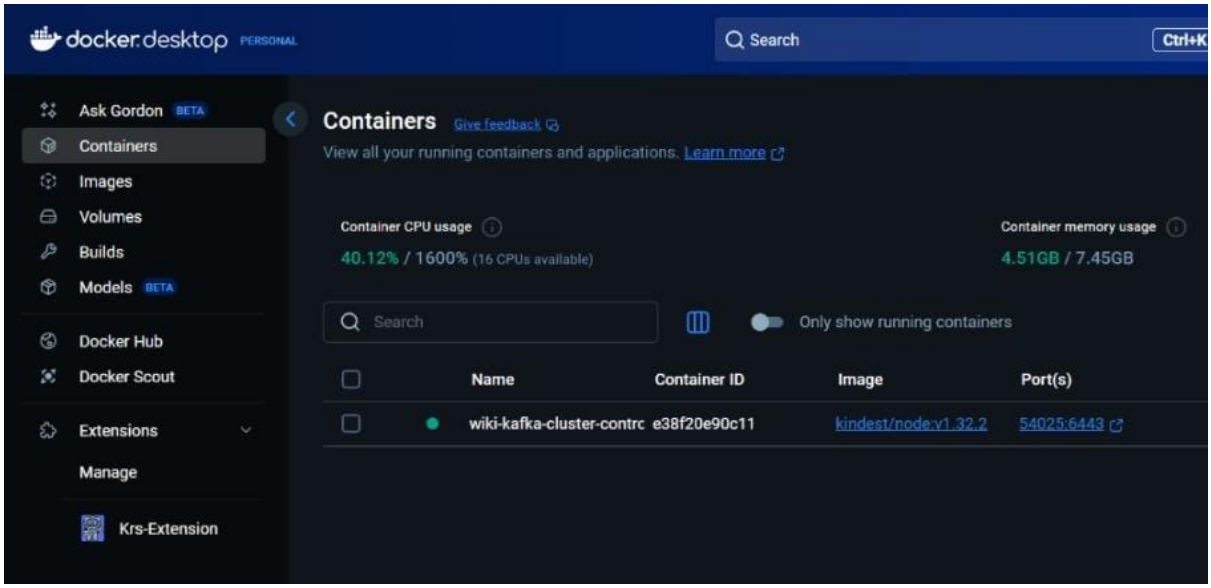


Figure 7: Docker Container Representation

Open Source / Third-Party Docker Images:

- **bitnami/elasticsearch-exporter:** Exposes Elasticsearch JVM, cluster, and indexing metrics to Prometheus.
- **docker.elastic.co/elasticsearch/elasticsearch:** Elasticsearch node for storing and querying data.
- **grafana/grafana:** Grafana UI for dashboard visualization.
- **confluentinc/cp-kafka-connect:** Runs Kafka Connect to link Kafka with Elasticsearch.
- **danielqsj/kafka-exporter:latest:** A Prometheus exporter that collects Kafka metrics like broker status, topic partitions, etc.
- **wurstmeister/kafka:** Runs the Apache Kafka broker to handle real-time message ingestion and publish-subscribe processing.
- **docker.elastic.co/kibana/kibana:** UI for visualizing data stored in Elasticsearch.
- **wurstmeister/zookeeper:** Provides coordination service for Kafka, managing broker metadata and leader election.
- **prom/prometheus:** Core Prometheus server for metrics scraping.
- **prom/node-exporter:** Exporter for host-level metrics (CPU, memory, etc.).

4.3 Kubernetes Deployment Setup

All pipeline components were deployed using `kubectl apply -f` commands. Key manifests included:

- `zookeeper.yaml` – Zookeeper Deployment and Service
- `kafka.yaml` – Kafka StatefulSet and Service
- `kafka-ui.yaml` – Kafka UI Deployment and Service
- `elasticsearch.yaml` – Elasticsearch StatefulSet and Service with volumes
- `kibana.yaml` – Kibana Deployment and Service
- `wiki-producer.yaml` – Wikipedia event producer Deployment and Service
- `wiki-processor.yaml` – Stream processor Deployment and Service
- `prometheus-deployment.yaml` – Prometheus setup with ConfigMap
- `grafana-deployment.yaml` – Grafana Deployment connected to Prometheus

StatefulSets ensured data persisted across restarts, leveraging the Kubernetes cluster's reliable volume handling.

The pipeline components were deployed using Kubernetes manifest files applied via `kubectl apply -f`. These manifests encompass Deployments, StatefulSets, Services, and ConfigMaps, which collectively facilitate the orchestration and management of the system's various components. StatefulSets are employed for Kafka and Elasticsearch to guarantee data persistence across pod restarts by utilizing persistent volumes provisioned by the Kubernetes cluster. Each manifest defines a critical service within the pipeline, including Zookeeper, Kafka, Kafka UI, Elasticsearch, Kibana, the custom Wikipedia event producer and processor, as well as Prometheus and Grafana for monitoring and visualization purposes.

Additionally, dedicated Kubernetes namespaces were created for organizing and isolating the components of the pipeline. This provided better manageability, security, and clarity in resource allocation when viewing pod statuses and other cluster resources.

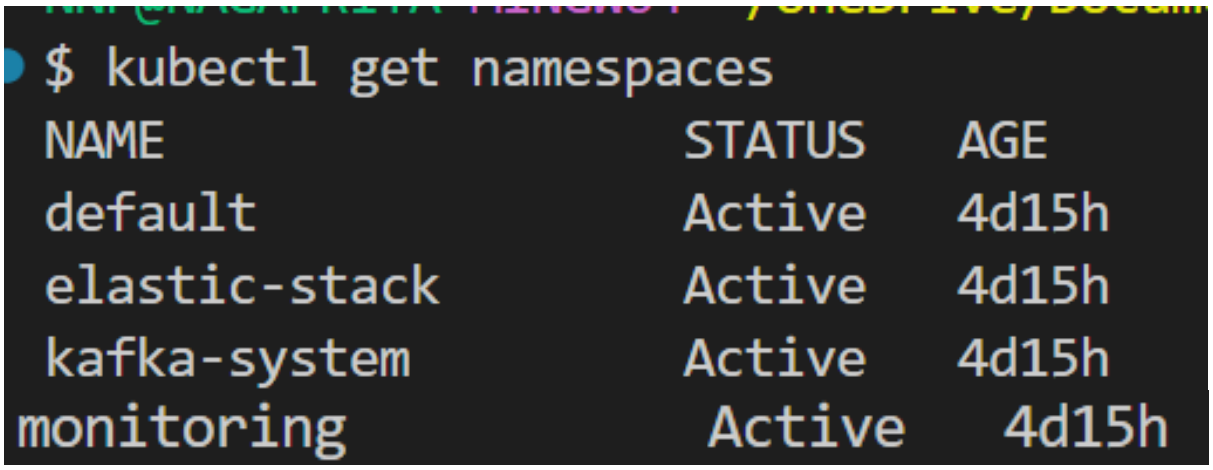


Figure 8: Kubernetes namespace list showing dedicated namespaces for pipeline components

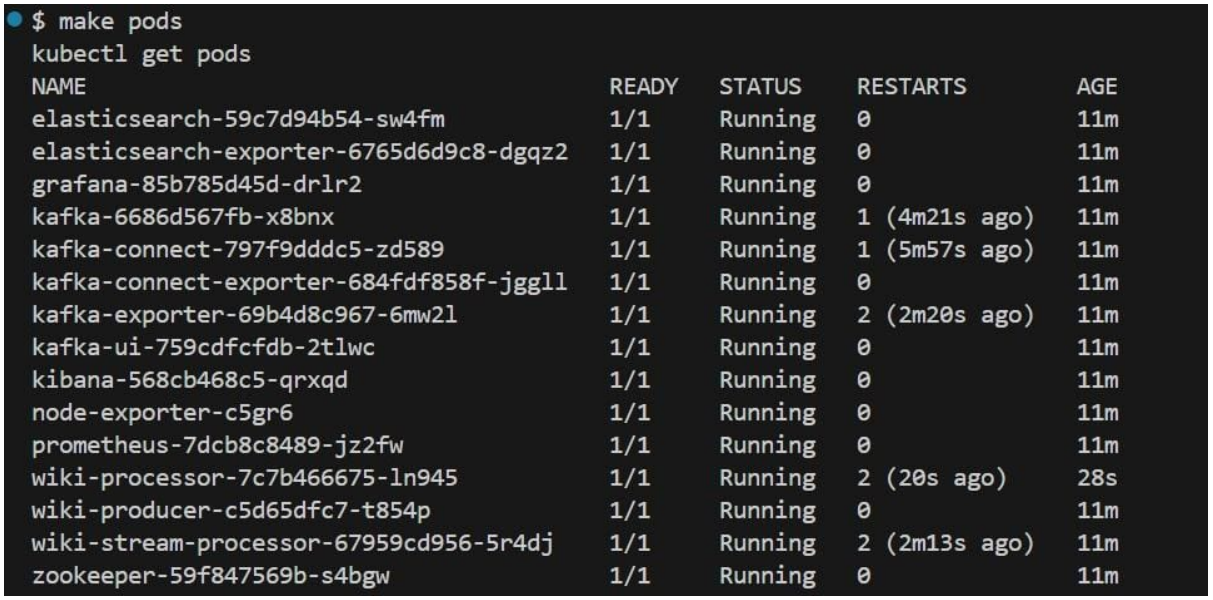


Figure 9: Running Pods in Kubernetes Cluster

Kibana: A visualization interface for Elasticsearch that allows users to explore, search, and analyze indexed data through dashboards and charts.

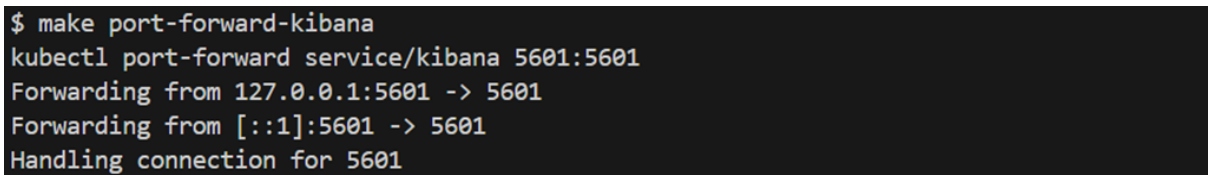


Figure 10: Setting up port forwarding to access Kibana

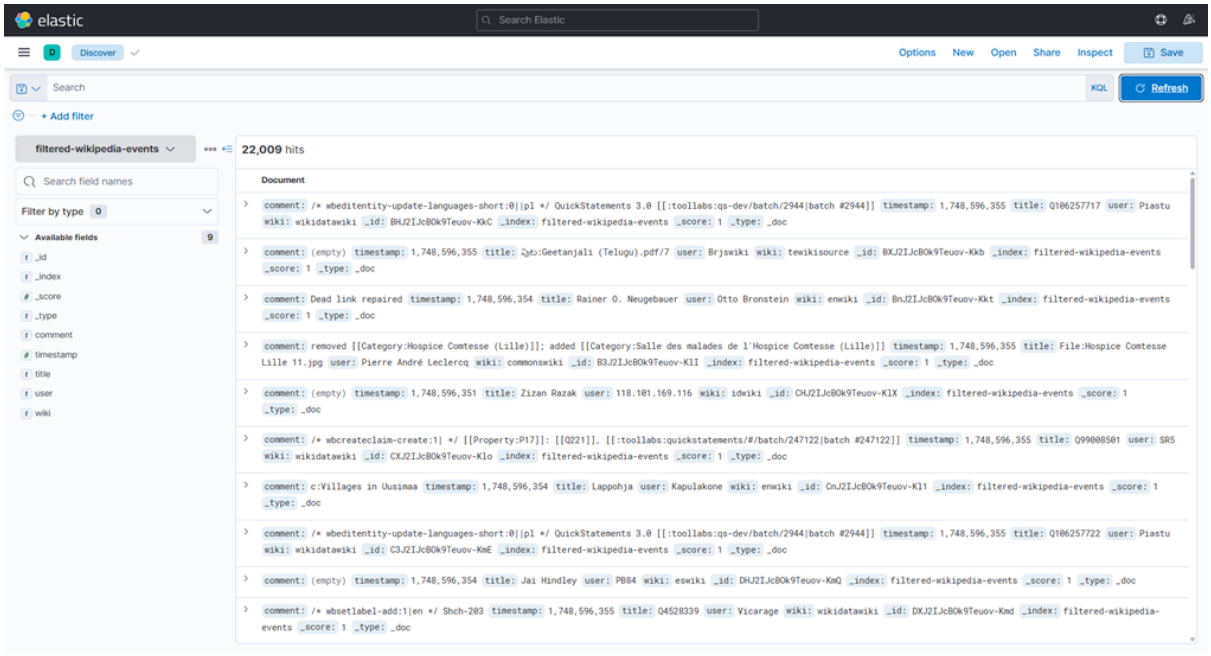


Figure 11: Visualizations of Filtered-Wikipedia-Events in dashboard

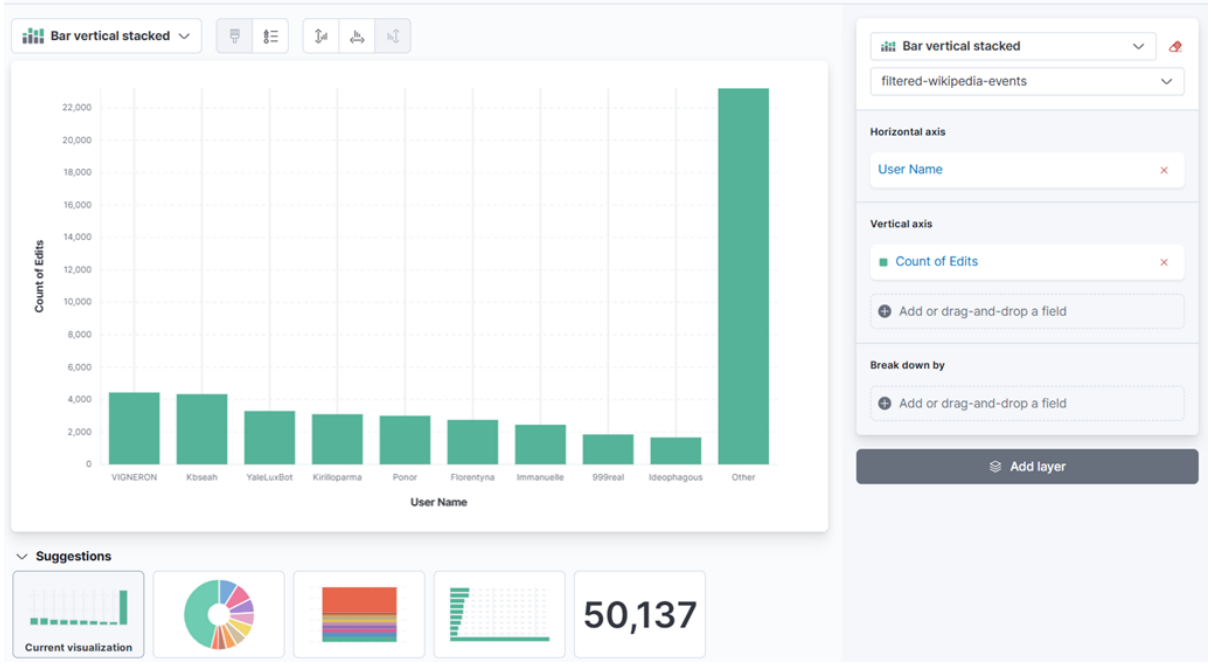


Figure 12: Visualization showing users with the highest number of edits in the past month.

5 Prometheus Deployment & Configuration

5.1 Prometheus ConfigFiles and ConfigMap

To enable robust observability for the real-time Wikipedia data streaming pipeline, Prometheus was deployed within the existing Kubernetes-in-Docker (KIND) cluster using a dedicated Kubernetes manifest file: `prometheus-deployment.yaml`. This file defined a `Deployment` resource referencing the official Prometheus Docker image and configured necessary volumes, ports, and service exposure.

A crucial aspect of the deployment was the use of a Kubernetes `ConfigMap` to manage the Prometheus configuration (`prometheus.yml`). Instead of embedding the configuration directly within the container image or using a `hostPath` mount, the `ConfigMap` method was adopted to enable:

- **Flexibility:** Allowing configuration updates without rebuilding the Docker image.
- **Version Control:** Enabling storage of the configuration alongside Kubernetes manifests in Git.
- **Separation of Concerns:** Decoupling Prometheus binary execution from its runtime environment-specific configuration.

The `ConfigMap` was mounted into the Prometheus Pod via a volume and referenced at Prometheus's default configuration path, `/etc/prometheus/prometheus.yml`. This specified:

- A global `scrape_interval` and `evaluation_interval` of 15 seconds.
- A list of static scrape targets for key services across the data pipeline, each defined under individual `job_name` blocks.

5.2 Prometheus Data Metrics

The core of Prometheus's monitoring capability lies in its `scrape_configuration`, defined in the `prometheus.yml` file under the `scrape_configs` section. In this setup, static targets were defined for all critical components of the pipeline, ensuring that Prometheus could fetch metrics from a known list of endpoints.

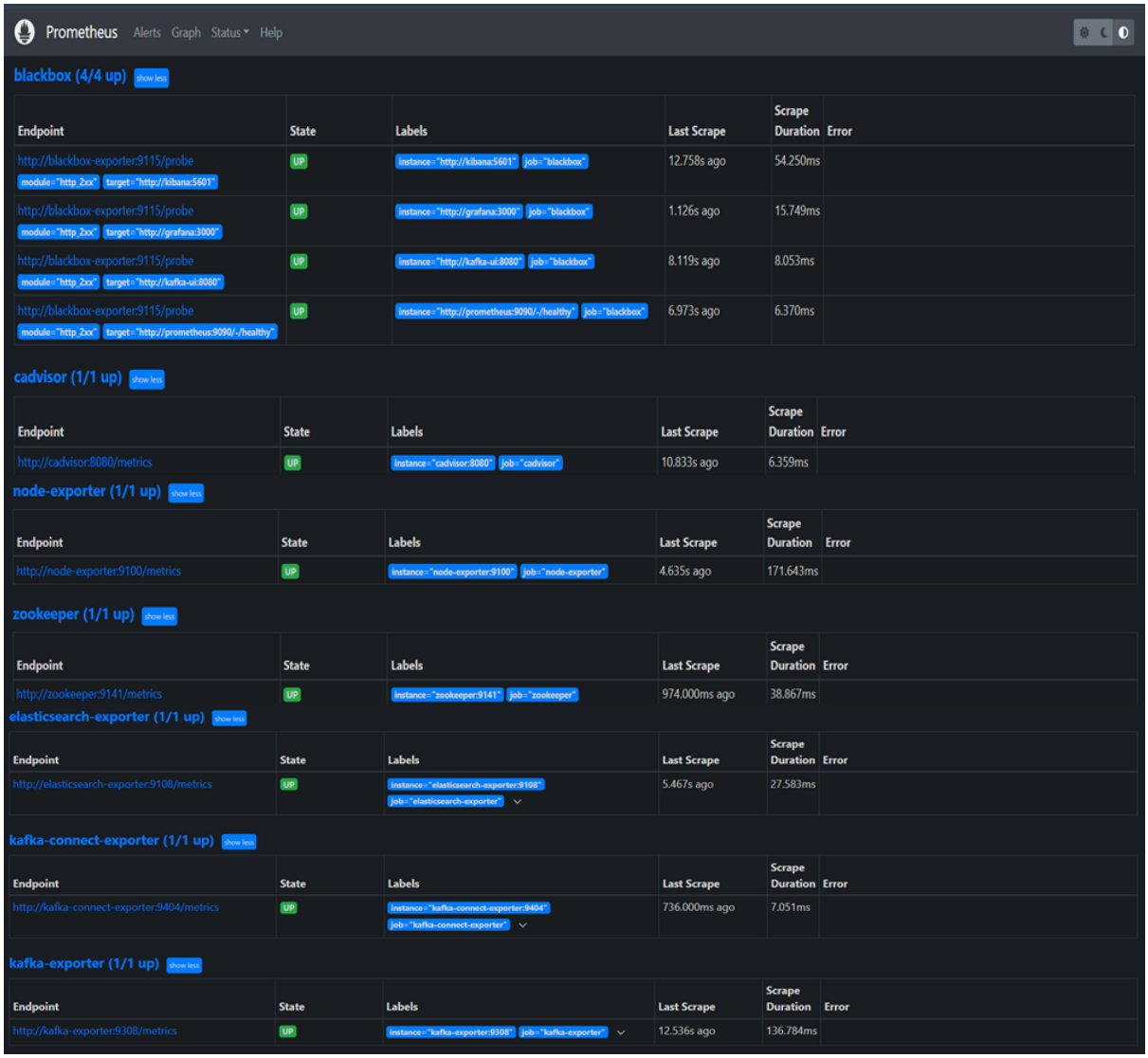


Figure 13: Prometheus dashboard showing all target jobs are up and successfully scraped.

Metrics:

- **Kibana:** validates kibana's HTTP endpoint using status checks
- **Grafana:** monitors grafana's availability and uptime
- **Kafka-ui:** probes kafka-ui web interface for responsiveness and accessibility
- **Prometheus:** ensure its own health and scrape functionality
- **Zookeeper:** tracks request latency, connections and health
- **Node-exporter:** provides system-level metrics like CPU, memory and disk usage from host
- **Cadvisor:** collects real-time container resource usage and performance metrics
- **Kafka-exporter:** Exposes Kafka metrics such as consumer lag, topic count, and broker health.
- **Elasticsearch-exporter:** Exports Elasticsearch stats like cluster health, indexing rate, and JVM metrics.

6 Prometheus Metrics Validation

To ensure Prometheus is successfully collecting metrics from all components of the wiki-streaming pipeline, a two-step validation approach was followed: inspecting target status and confirming real-time scraping activity via the Prometheus web UI.

6.1 Target Status in Prometheus UI

The Targets tab in the Prometheus UI `http://<prometheus-service>:9090/targets` was used to verify the health and reachability of all metric endpoints. Each target corresponds to a job configured in the Prometheus scrape configuration defined in prometheus.yaml.

Component	**Job Label**	**Target URL**	**Port**
Kafka Exporter	kafka-exporter	kafka-exporter:9308	9308
Elasticsearch Exporter	elasticsearch	elasticsearch-exporter:9114	9114
Node Exporter (Kubernetes)	node-exporter	<node-ip>:9100	9100
Wiki Producer (Python App)	wiki-producer	wiki-producer:8000	8000
Wiki Processor (Python App)	wiki-processor	wiki-processor:8001	8001

Table 1: Monitoring Components and Target Endpoints

Each listed target showed:

- UP status (green checkmark).
- Last scrape duration well within configured scrape interval (typically 15s or 30s).
- No error or timeout messages.

This confirmed that Prometheus could reach and scrape all exporters and custom instrumented services correctly.

7 Grafana Visualization Setup

Grafana was deployed alongside Prometheus within the Kubernetes cluster to enable real-time visualization of pipeline metrics. It transformed raw time-series data from Prometheus into rich, insightful dashboards that supported system observability and proactive debugging.

7.1 Connecting Prometheus Data Source

The Grafana deployment was configured to connect to Prometheus as its primary data source. This setup was done through the Grafana UI:

- A new data source of type **Prometheus** was added.
- The data source URL was configured as `http://prometheus:9090`, assuming the Prometheus service in Kubernetes was named `prometheus` and exposed on port `9090`.

This connection allowed Grafana to query all metrics collected by Prometheus from various services in the pipeline.

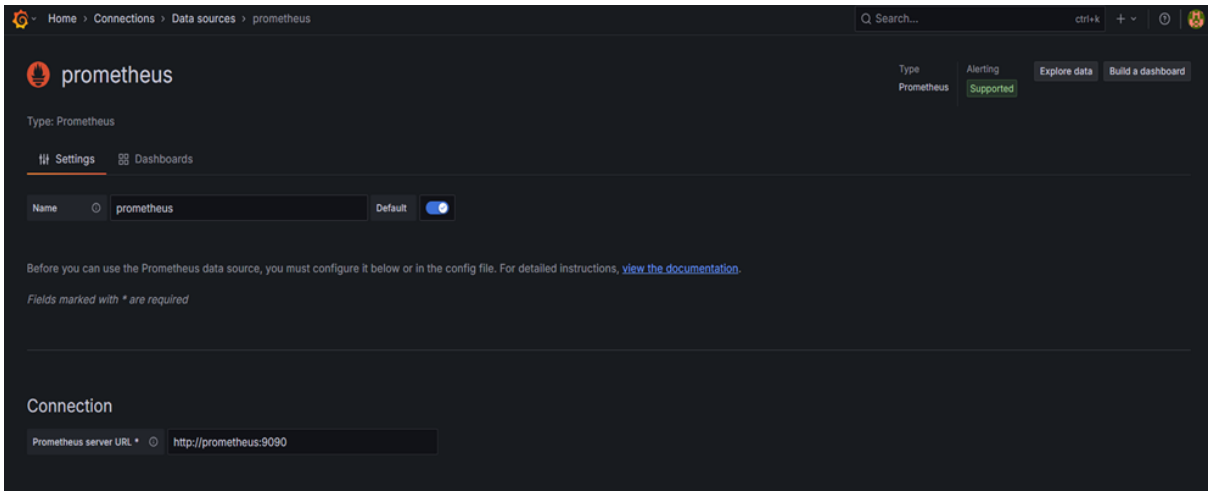


Figure 14: Setting Up Prometheus as a Data Source in Grafana

7.2 Grafana Visualization Dashboards

Elasticsearch Dashboard: Offered cluster health metrics, node resource utilization, indexing/search throughput, JVM internals, and much more. These prebuilt dashboards significantly reduced setup time and offered deep observability with minimal configuration.

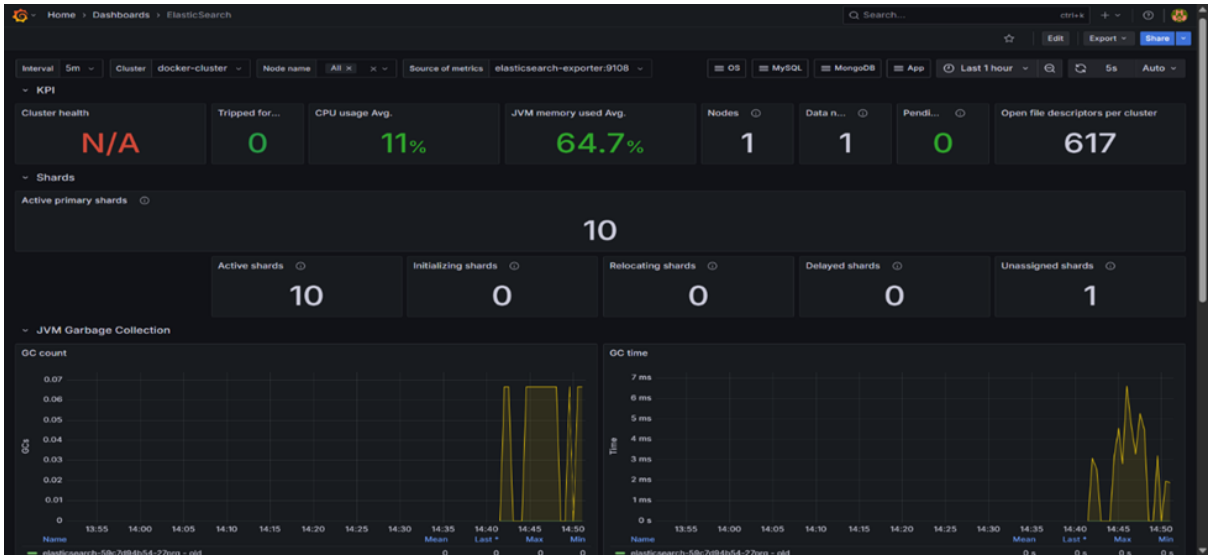


Figure 15: Elasticsearch Dashboard Visualization

Key Metrics:

- **Cluster health:** indicates overall health of Elasticsearch cluster
- **CPU usage average:** average CPU utilization of Elasticsearch nodes
- **JVM memory used average:** shows average JVM heap used for performance tuning
- **Nodes:** total nodes detected in cluster
- **Data nodes:** count of nodes holding data
- **Pending nodes:** tasks waiting in queue
- **Open file descriptors:** OS-level handlers used by Elasticsearch (high number indicates memory pressure)

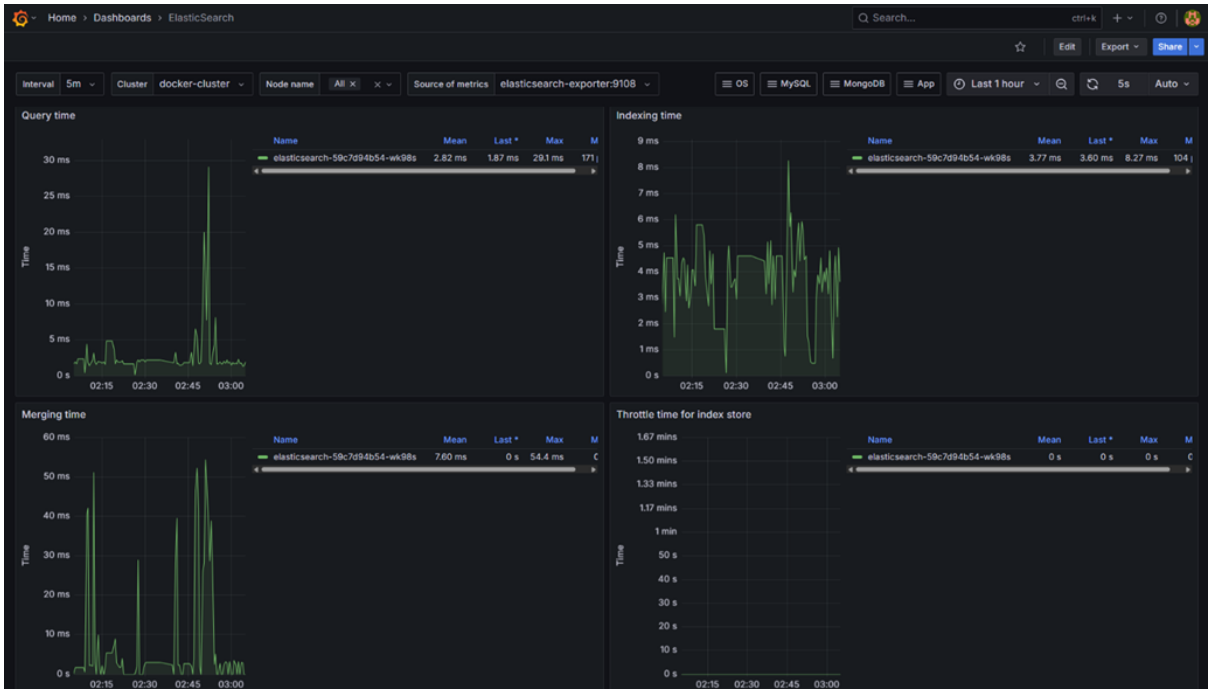


Figure 16: Elasticsearch Dashboard Displaying Time-Based Metrics

Key Metrics:

- **Query time:** measures time to serve search queries
- **Indexing time:** time taken to index new documents
- **Merging time:** time taken to merge index segments
- **Throttle time:** time elasticsearch spent throttling index operation to manage resource

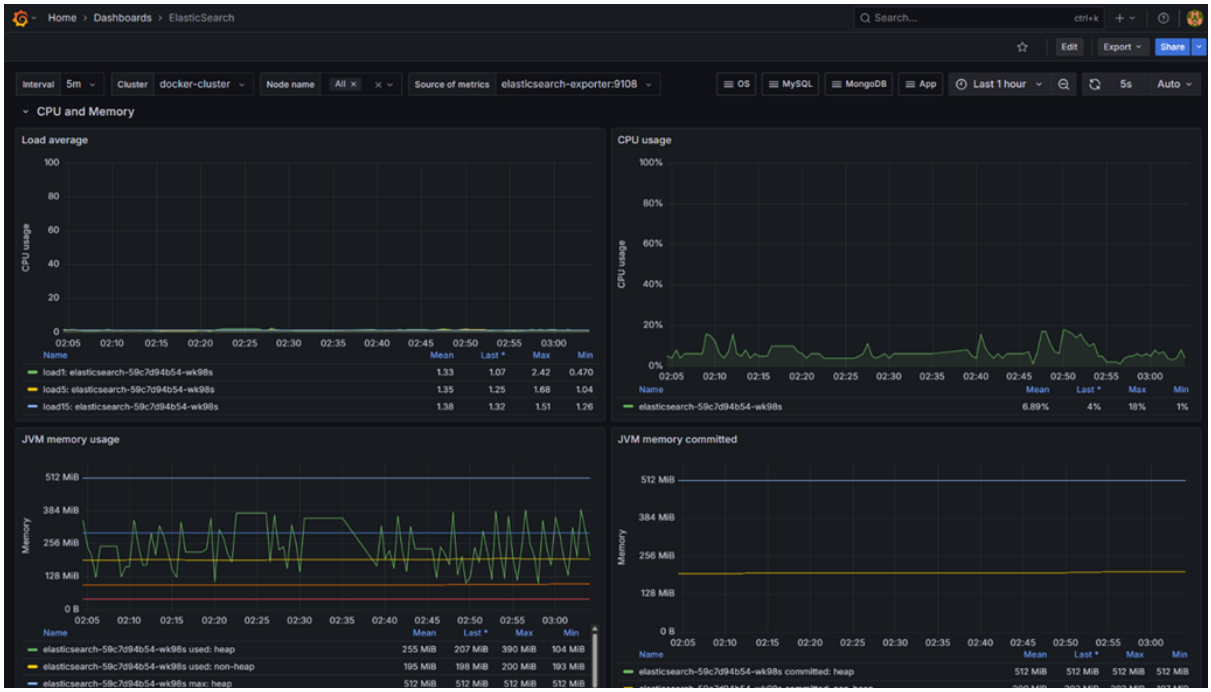


Figure 17: Elasticsearch Dashboard Displaying CPU and Memory Usage

Key Metrics:

- **Load average:** CPU load on nodes
- **CPU usage:** real time and historical CPU usage by elasticsearch
- **JVM memory usage:** tracks heap and non-heap memory usage
- **JVM memory committed:** slows total heap space reserved for JVM

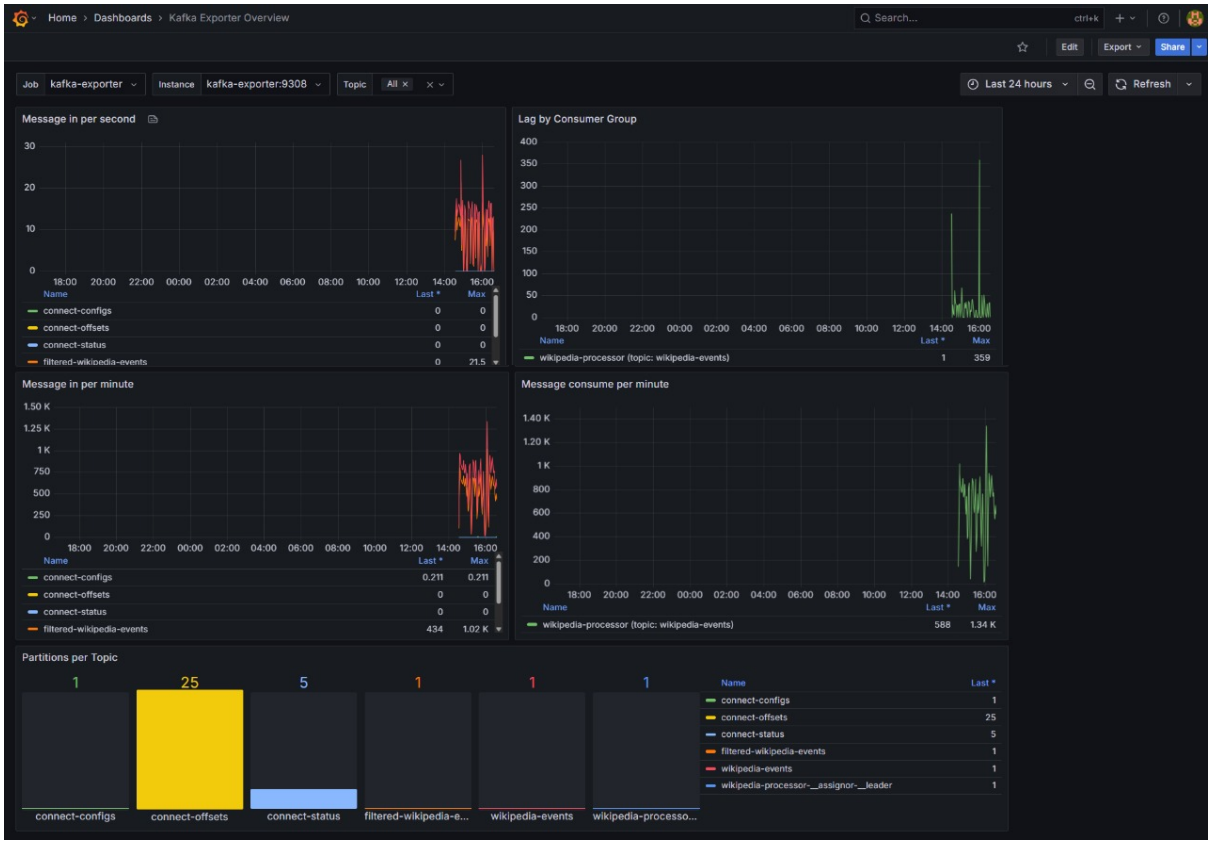


Figure 18: Kafka Dashboard Displaying Real-Time Metrics and Partitions

Key Metrics:

- **Messages In per Second:** Displays real-time incoming message rate for each Kafka topic.
- **Lag by Consumer Group:** Shows how delayed the consumer is from the latest messages.
- **Messages In per Minute:** Tracks the per-minute ingestion rate for all topics.
- **Messages Consumed per Minute:** Visualizes how many messages are being consumed per minute.
- **Partitions per Topic:** Indicates the number of partitions assigned to each Kafka topic.

8 Observability and Traceability

8.1 Monitoring using Prometheus

The foremost objective of the Prometheus integration was to monitor the health and availability of each pipeline component.

Prometheus was configured to scrape metrics from Kafka, Elasticsearch, and custom services like the Wikipedia producer and stream processor. These metrics include:

- JVM memory usage and garbage collection time
- Kafka topic throughput (messages in per second)
- Elasticsearch cluster health, CPU usage, node counts, and heap memory
- Exporter data from Kafka and Elasticsearch nodes

The collected metrics were visualized in Grafana dashboards, enabling real-time analysis of system performance and resource usage.

8.2 Logging & Traceability

Each service logs key events (e.g., message production, consumption, errors). Kubernetes `kubectl logs` was used to trace the flow of messages from the producer, through Kafka, and into Elasticsearch.

wiki-producer: A Python service that streams live Wikipedia edit events and publishes them to the Kafka topic `wikiTopic`.

```
$ make producer-logs
kubectl logs deployment/wiki-producer -f
✅ Connected to Kafka!
🔄 Listening to Wikipedia events...
📤Sending to Kafka: Diskussion:Valerii Kryushyn edited by 2A00:20:73BD:E419:F294:C19E:68D2:1FC1
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7326
📤Sending to Kafka: Науан Досмағамбетов edited by Орел Карл
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7327
📤Sending to Kafka: Света гора (квартал) edited by 130.204.252.57
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7328
📤Sending to Kafka: Q28203309 edited by VIGNERON
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7329
📤Sending to Kafka: File:StateLibQld 1 200591 Brunnich family posing on and around the front stairs of their home at Gatton, Queen
sland, 1900.jpg edited by Chris.sherlock2
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7330
📤Sending to Kafka: Njohuritë mediatike edited by Edukatori
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7331
📤Sending to Kafka: File:NDL9892820 闇思獣境界 - 2巻.pdf edited by MidleadingBot
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7332
📤Sending to Kafka: File:Alstom Coradia Continental Enno Innotrans 2014.JPG edited by SchlurcherBot
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7333
📤Sending to Kafka: Onwuasoanya edited by Aminwa 21
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7334
📤Sending to Kafka: Q134880540 edited by Putnik
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7335
📤Sending to Kafka: Q28951559 edited by VIGNERON
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7336
📤Sending to Kafka: Q5397710 edited by Ideophagous
✅ Message sent: wikipedia-events | Partition: 0 | Offset: 7337
```

Figure 19: Logs of the `wiki-producer` Pod Captured via `kubectl logs`

wiki-processor: A Faust-based stream processing service that consumes events from Kafka, filters or transforms them, and sinks the results into Elasticsearch.

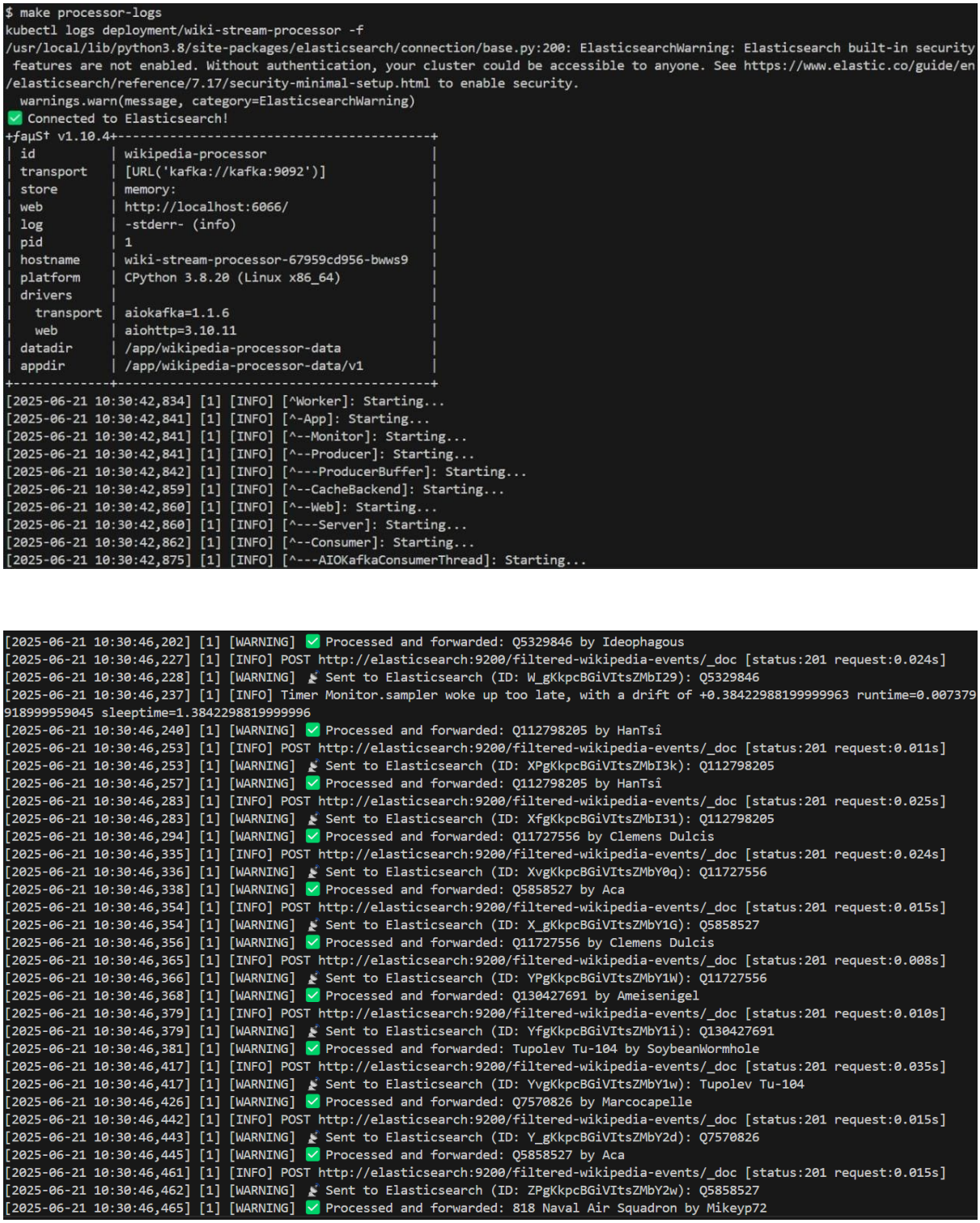


Figure 20: Logs of the `wiki-stream-processor` Pod Captured via `kubectl logs`

Functional Dataflow:

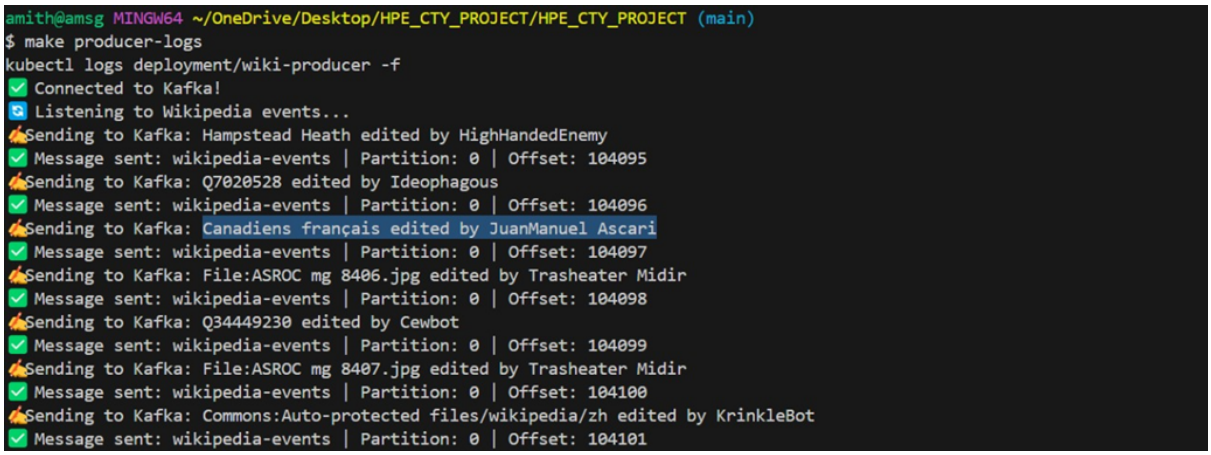


Figure 21: Kafka producer logs showing a real-time Wikipedia edit event being published.

This log output shows a real-time Wikipedia edit event (e.g., `"Canadiens français edited by JuanManuel Ascari"`) being produced by the Kafka producer and sent to the Kafka topic named `wikipedia-events`.

Kafka UI: A user-friendly web interface to monitor and manage Kafka clusters, topics, and consumer groups.

Figure 22: Setting up port forwarding to access Kafka-ui

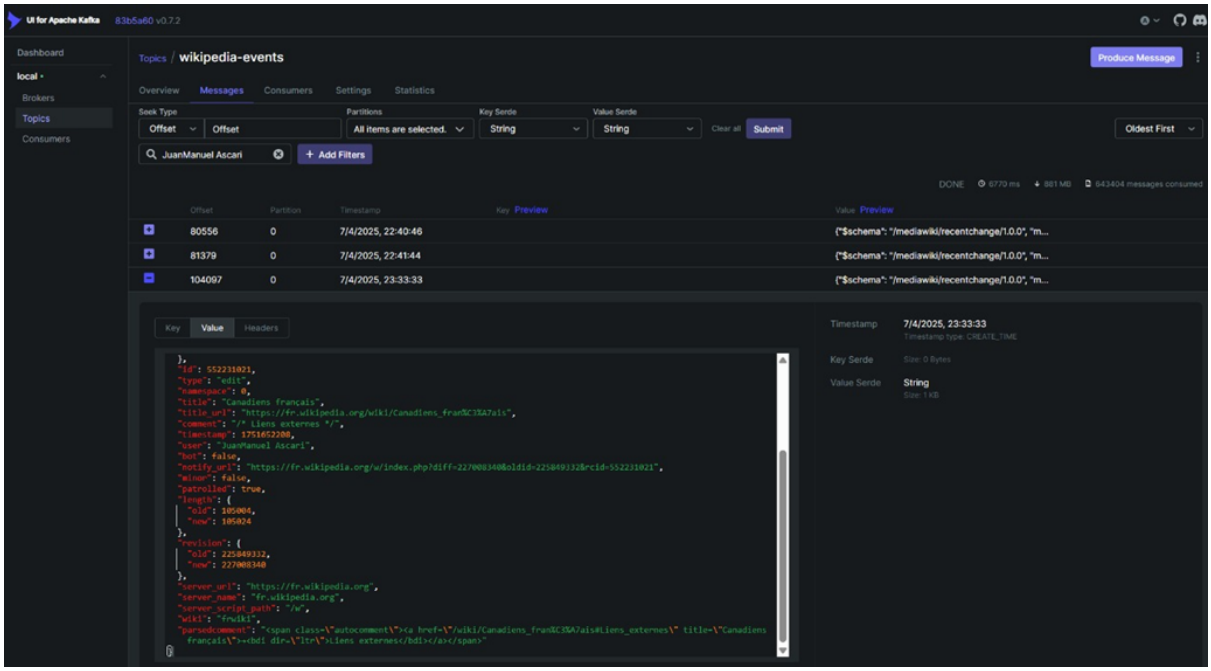


Figure 23: : Kafka-ui dashboard showing message produced in wikipedia-events topic

The same message is displayed in the Kafka UI under the topic filtered-wikipedia-events, confirming that the message was successfully published and processed.

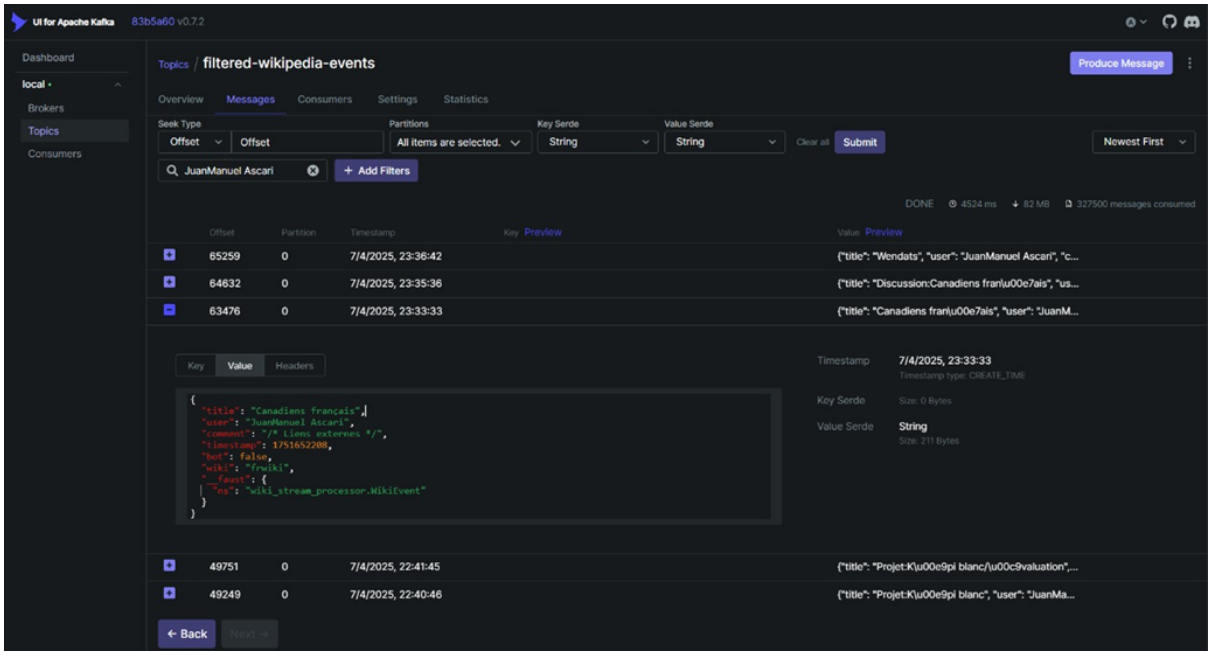


Figure 24: Kafka UI confirming the message was successfully sent to the filtered-wikipedia-events topic.

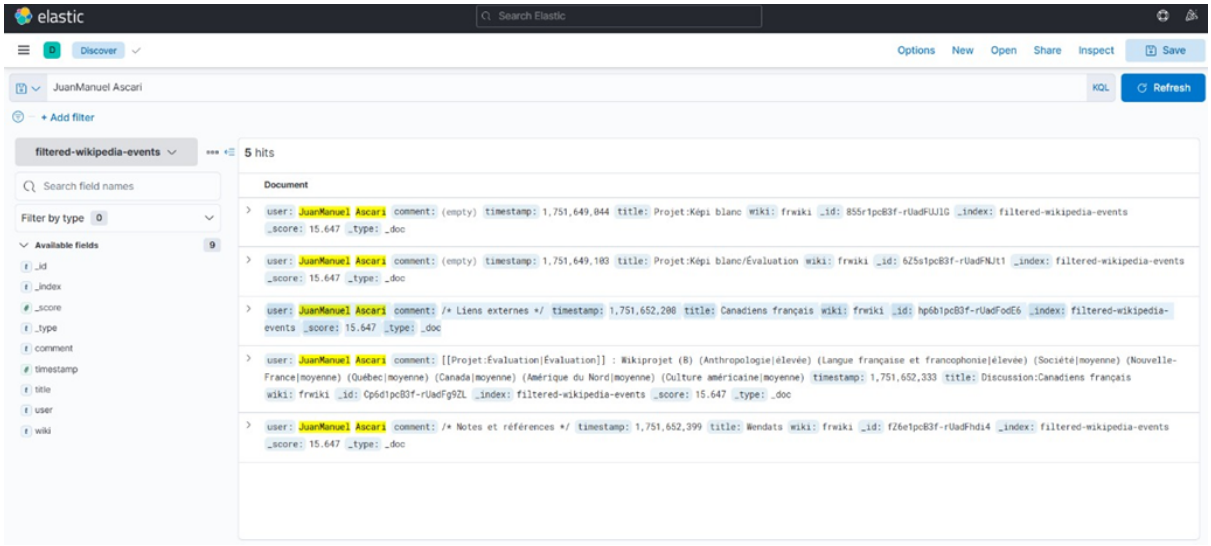


Figure 25: Kibana displaying the stored event in Elasticsearch, completing the data pipeline.

This snapshot shows the same message stored in Elasticsearch, searchable through Kibana, completing the data flow from Wikipedia to Elasticsearch via Kafka.

The Kafka UI helped visualize the messages within topics, and Kibana was used to verify data ingestion in Elasticsearch, providing end-to-end traceability of the pipeline.

8.3 Alert System Using SMTP

To proactively respond to system issues, alerting mechanisms were established using Prometheus Alertmanager. Alerts were configured based on predefined thresholds, such as:

- Kafka or Elasticsearch pod failures
- Excessive memory or CPU usage
- Service unavailability

These alerts were delivered via SMTP email notifications, ensuring that the development team was immediately informed of any critical failures or performance degradation.

Alert Rule: Elasticsearch JVM Heap $> 80\%$

SMTP (Simple Mail Transfer Protocol) is a communication protocol used to send email messages between servers. Here, SMTP is used to trigger and deliver alert emails when specific conditions are met (e.g., JVM heap memory usage $> 80\%$), enabling automated notifications as part of the monitoring and alerting system.

- **Purpose:** Detects high JVM heap memory usage in Elasticsearch nodes to prevent memory pressure issues.
- **Condition:** Heap usage > 80% for a continuous 1 minute (pending period).
- **Severity:** `warning`. Labeled for the DevOps team to take action.

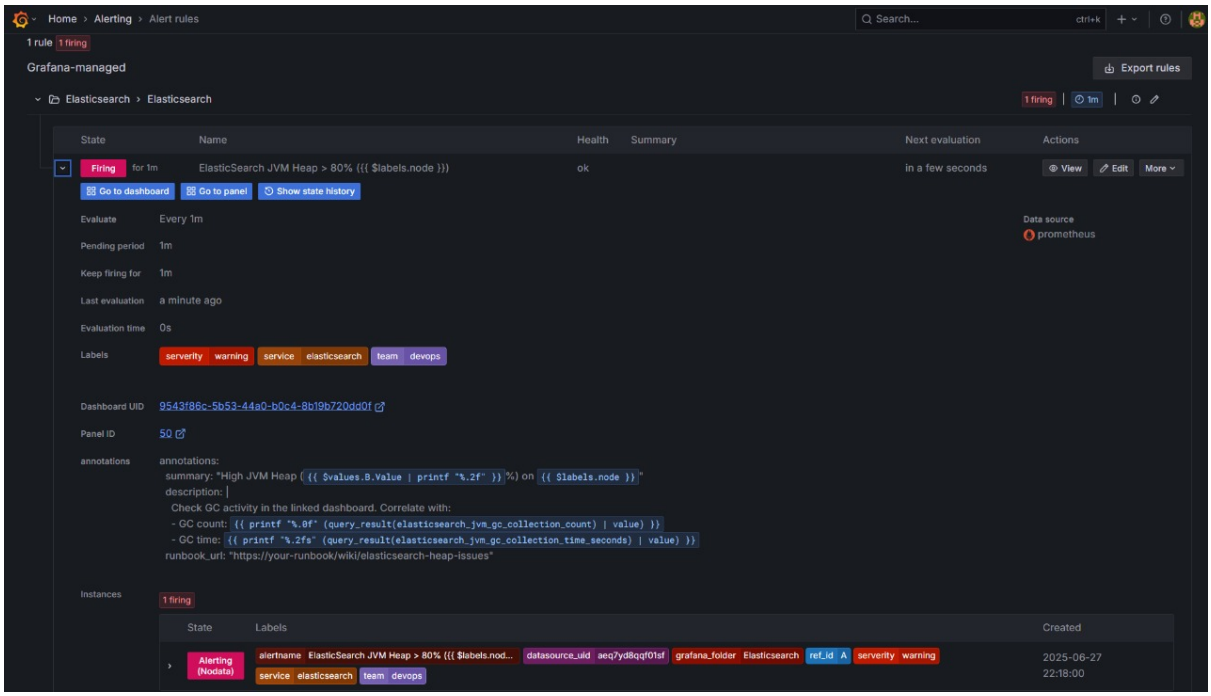
Notifications

- **Current Contact Point:** `elasticsearch-alerts-email`
- **Silencing:** Alerts can be silenced for maintenance via Grafana → Alerting → Silences

Dashboard Integration

The monitoring system was integrated with Grafana dashboards for real-time visualization of key metrics. Specific panels included:

- Integrated with Elasticsearch Grafana dashboard panels: JVM Memory Used Avg.



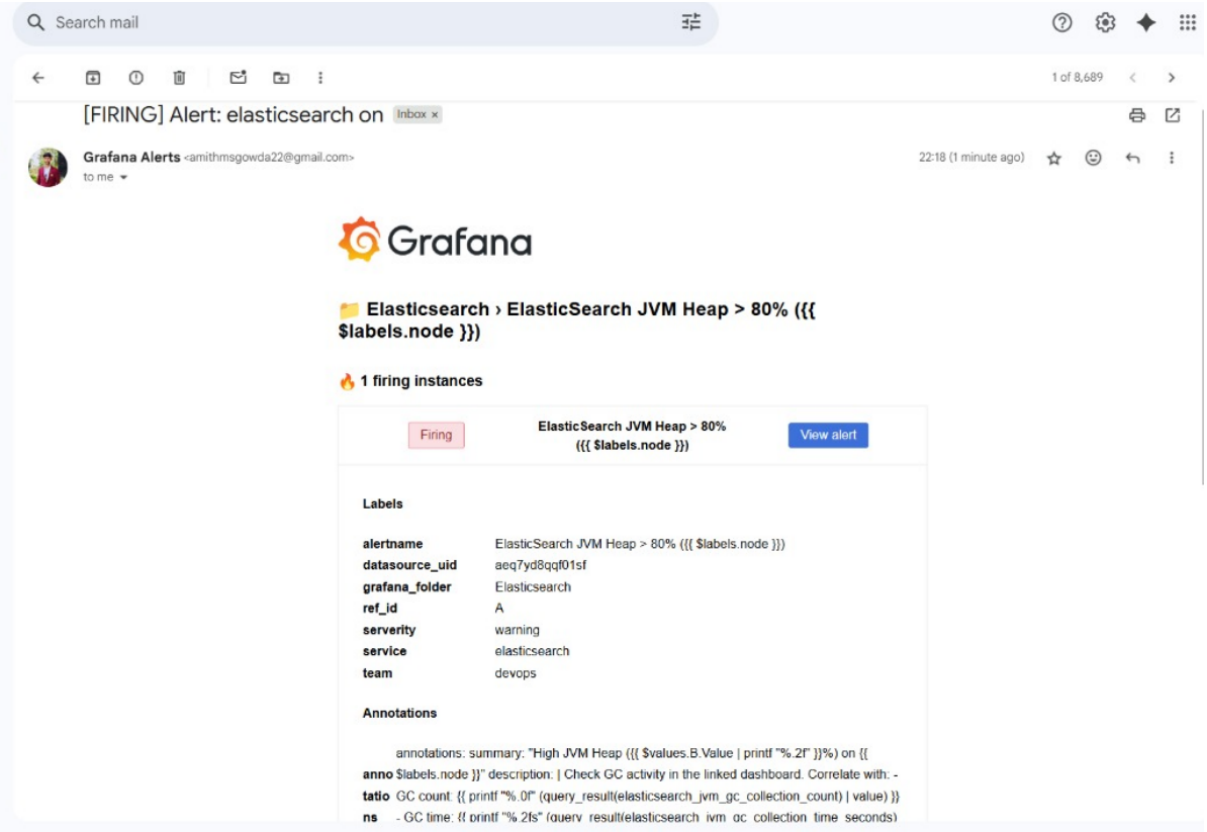


Figure 26: Email notification showing an active (firing) alert status.

9 Conclusion & References

This project successfully designed and implemented a scalable, fault-tolerant real-time data streaming pipeline from Wikipedia events ingested via Kafka, processed with custom Python services, and indexed in Elasticsearch using Kafka Connect. The entire system is containerized and orchestrated with Kubernetes (KIND), with robust monitoring provided by Prometheus and Grafana.

Key achievements:

- Reliable multi-node Kubernetes cluster deployment using KIND
- Full pipeline integration from source ingestion to searchable storage
- Comprehensive monitoring and alerting with Prometheus and Grafana
- Modular and containerized architecture supporting easy extensibility

References:

- **Apache Kafka Documentation:** Apache Kafka reference
- **Elasticsearch Documentation:** ElasticSearch reference
- **Grafana Documentation:** Grafana reference
- **Prometheus Documentation:** Prometheus reference
- **Phase 1 GitHub:** Wiki-Project (Docker Desktop Kubernetes)
- **Phase 2 GitHub:** Wiki-Project (Minikube integration)
- **Phase 3 GitHub:** Wiki-Project (KIND implementation)
- **Monitoring Implementation Phase:** GitHub
- **Final Wikipedia-Streaming Project:** GitHub

10 Project Timeline

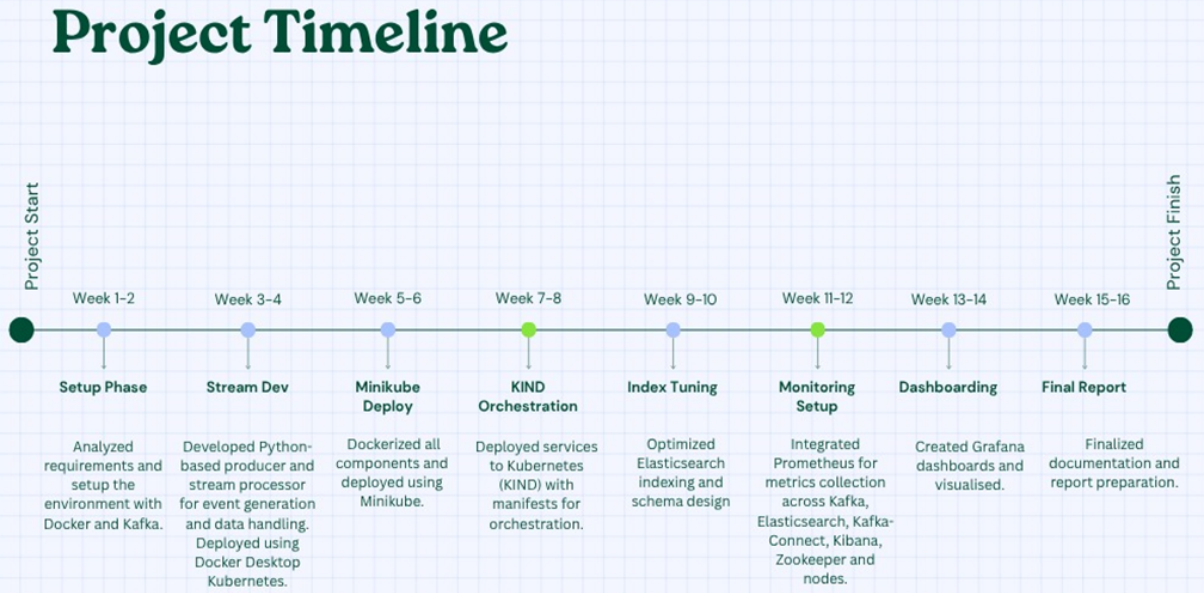


Figure 27: Project Timeline Representation.