Cloud Computing Assignment Report

Student: Mustafa Fahimy

Course: Cloud Computing (WS 2024/25)

University of Vienna

1. Introduction

This project implements and deploys an event-driven consumer service as part of the Leaf Image Management System (LIMS). The system leverages Apache Kafka, MongoDB, and Kubernetes, and is deployed both locally (Minikube) and on Google Kubernetes Engine (GKE). The objective is to compare stream processing with batch processing in terms of performance and bandwidth usage, using Prometheus and Grafana for observability.

2. Implementation Details

2.1 System Setup

The base system includes:

- **Image API** (stream processing)
- **Kafka cluster**
- **Producer and Consumer MongoDB databases**
- **Camera job** (image generation)
- **Leaf Disease Recognizer job**
- **DB Synchronizer job** (batch processing)

Deployment:

- Locally via .stg manifests (start.stg.sh).
- On GKE via .prod manifests (start.prod.sh).

2.2 Kafka-based Consumer Service

- Developed a custom **Kafka consumer** in Python.
- Subscribed to Kafka topics for leaf images (potato).
- Consumed events in real time and stored them in **Consumer MongoDB**.
- Resource constraints applied in consumer-deployment.yaml (CPU/memory limits).

  The consumer follows an **event-driven architecture**, reacting to Kafka events instead of polling or batch execution.

2.3 HTTP API for Data Access

To meet Task 2 requirements, the consumer exposes REST endpoints via FastAPI:

- GET /image-plant/{image_type}/{image_id} → retrieve image by ID.
- GET /image-plant/{image_type}/total → retrieve total image count.

Ingress routes these endpoints as the **only external entry point** on GKE.

This satisfies the assignment requirement that **Ingress is the only external entry point** on GKE.

3. Kubernetes Objects and Service Types

3.1 Objects Used

- **Deployment** → manages consumer pods and scaling.
- **Service (ClusterIP)** → internal communication.
- **Ingress** → external entry point on GKE.
- **ConfigMap** → stores Kafka/MongoDB connection details.

3.2 Service Types

Service Type	Usage
ClusterIP	Internal communication (MongoDB, Kafka)
NodePort	Pre-implemented services (Image API, DB Synchronizer)
LoadBalancer	Grafana external access on GKE
Ingress	Consumer endpoints exposed externally

4. Challenges Faced

- Docker image build issues due to local daemon configuration.
- Kafka connectivity errors (bootstrap server misconfiguration).
- Python dependency conflicts (bson vs pymongo).
- Switching contexts between Minikube and GKE.
- Understanding ingress-only exposure requirements on GKE.

Resolved through configuration validation, pod log inspection, and iterative debugging.

5. Scalability, Bottlenecks, and Architecture Analysis

5.1 Is the Service Scalable?

Yes:

- Increase Deployment replicas.
- Kafka supports multiple consumers in a group.

- MongoDB supports sharding and replica sets.

5.2 Increased Dataflow Impact

- Consumers scale horizontally.
- Kafka partitions enable parallelism.
- MongoDB write throughput may bottleneck without scaling.

5.3 Batch vs Stream Processing

- **Batch** → periodic spikes, latency, uneven bandwidth.
- **Stream** → continuous, low-latency, smoother resource usage.
- Stream is superior for real-time workloads.

5.4 Improving Batch Processing

- Reduce batch intervals.
- Use incremental updates.
- Parallelize batch jobs. Still less efficient than stream processing.

6. Scaling Strategy on GKE

- Horizontal Pod Autoscaler (HPA) for consumer pods.
- Increase Kafka partitions.
- Scale MongoDB replicas.
- Use managed services for resilience.

7. Deployment Modifications for GKE

- Used .prod manifests.
- Configured Ingress as sole external entry point.
- Pushed Docker images to Google Artifact Registry.
- Adjusted service types (NodePort → LoadBalancer/Ingress).

8. GKE Cluster Configuration

- **Cluster type**: GKE Standard.
- **Region**: europe-west4-a.
- **Node pool**: Default.
- **Machine type**: e2-standard-4 (6 vCPUs, 24 GB RAM).
- **Autoscaling**: Enabled.

9.1 Batch vs Stream Cost

- **Batch** → requires larger nodes to handle spikes.
- **Stream** → smoother load, smaller nodes suffice.
- Rough estimate:
    - Batch: ~€1,200/year.
    - Stream: ~€800/year.

9.2 Cloud vs On-Premise

- Cloud avoids upfront hardware costs.
- On-premise > €5,000/year (hardware, maintenance, power).
- Cloud is more cost-effective for academic workloads.

10. Resource Justification

- **Consumer**: 250–500m CPU, 256–512Mi memory.
- **Kafka broker**: ~1 vCPU, 2 GB RAM.
- **MongoDB**: 2 vCPU, 4 GB RAM.
- Balanced for performance and cost efficiency.

11. Metrics and Performance Comparison (Task 3)

- Prometheus deployed via start-metrics.prod.sh.
- Grafana dashboards imported.
- Observations:
    - **Stream (Image API)** → smooth, continuous bandwidth.
    - **Batch (DB Synchronizer)** → stepwise spikes.
- Confirms stream processing efficiency advantage.

Stream Processing (Image API)

- **Metric**: image_api_image, image_api_image_size
- **Pattern**: Smooth, continuous increase in image count and size over time.
- **Interpretation**: Stream processing handles data incrementally and consistently, ideal for real-time updates.

Batch Processing (DB Synchronizer)

- **Metric**: db_synchronizer_job_image, db_synchronizer_job_image_size

- **Pattern**: Stepwise jumps in image count, with flat periods in between.
- **Interpretation**: Batch processing transmits data in chunks, leading to latency and uneven bandwidth usage.

Final Insight

Stream processing offers lower latency and more consistent bandwidth usage, making it better suited for real-time applications like image ingestion and metadata updates. Batch processing, while simpler, introduces delays and bandwidth spikes that can affect responsiveness and scalability.

12. Conclusion

This assignment demonstrated:

- Implementing a Kafka consumer locally and on GKE.
- Configuring Ingress for secure external access.
- Deploying Prometheus + Grafana for observability.
- Comparing batch vs stream processing bandwidth.
- Analyzing scalability, costs, and Kubernetes object usage.

Key takeaway: Stream processing provides superior efficiency and responsiveness compared to batch processing, especially in cloud-native, event-driven architectures.

13. Screenshots

The following screenshots are included:

• GKE cluster overview

• Workloads (Pods and Deployments)

• Services and Ingress configuration

• Grafana dashboards comparing batch and stream processing

Figure 1 shows the cluster

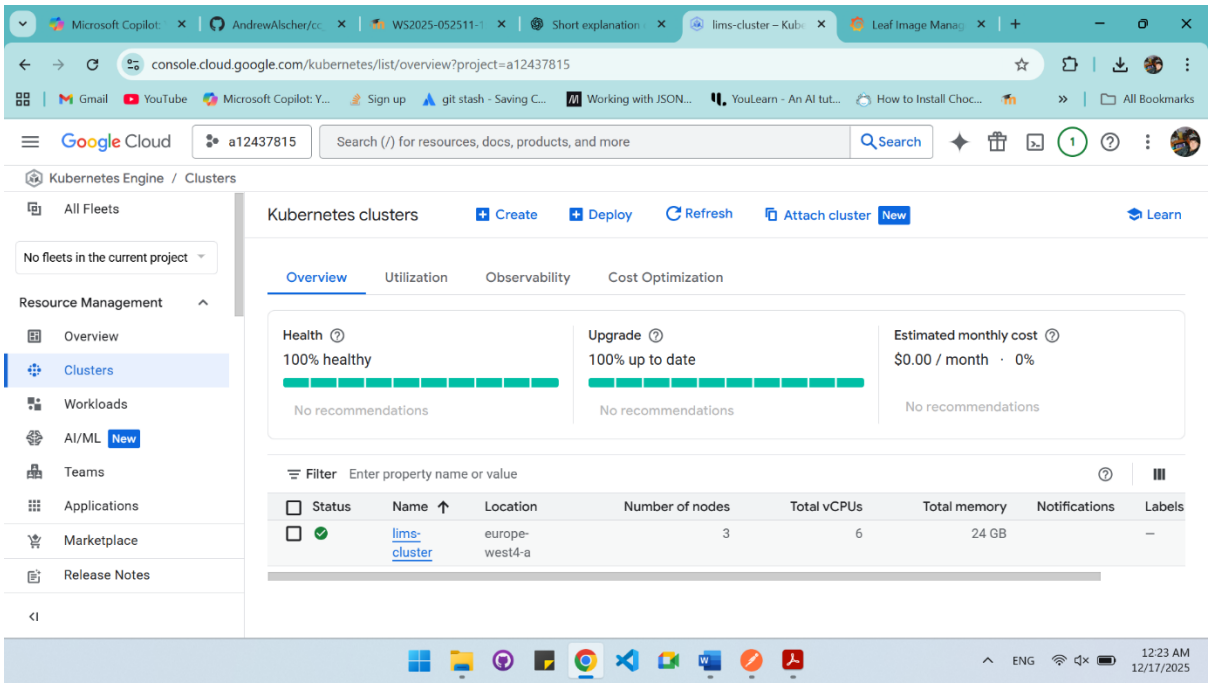

Figure 1 shows external test status

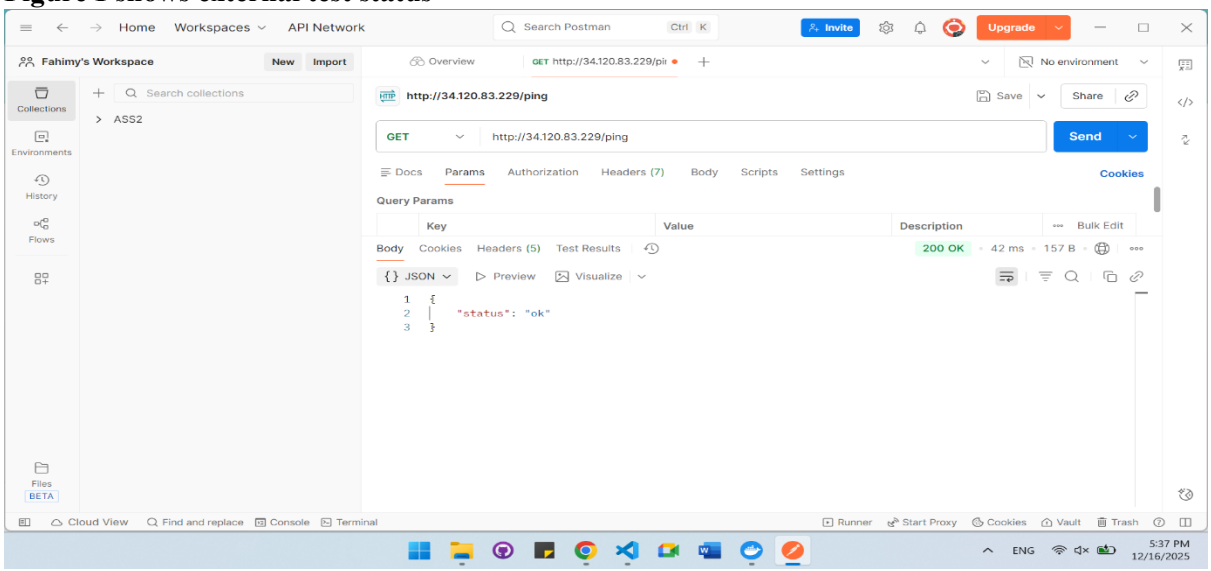

Figure 1 shows total records

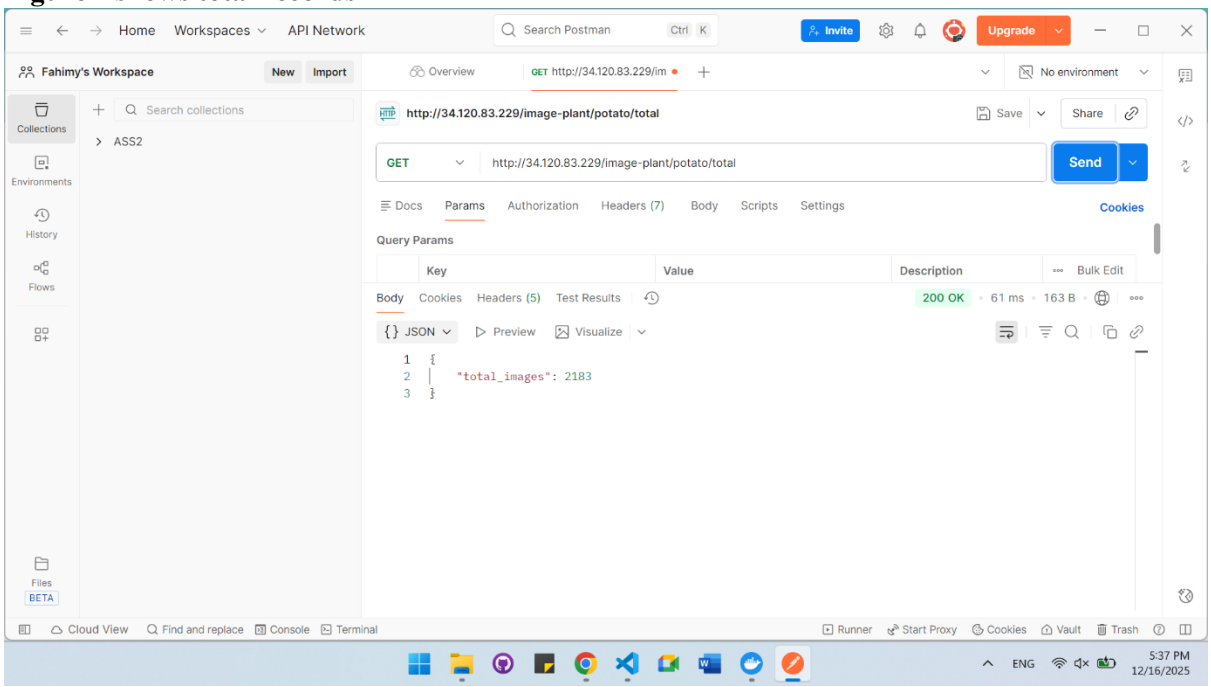

Figure 1 shows one record

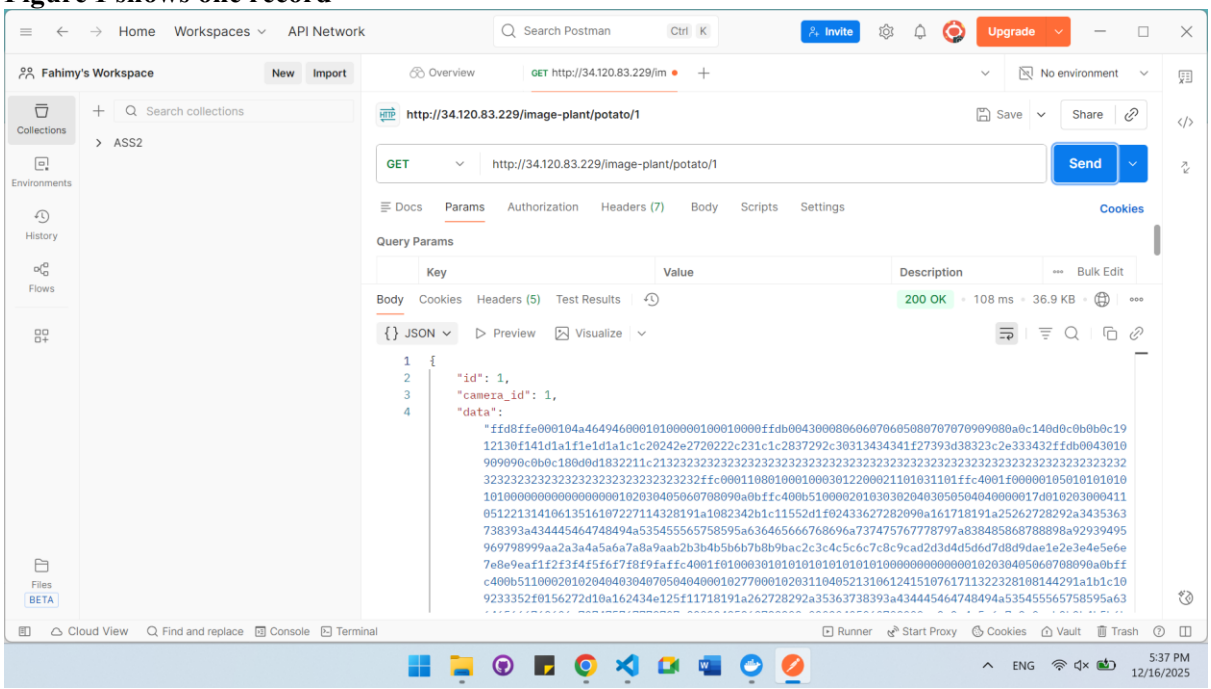

Figure 1 shows Grafana dashboard

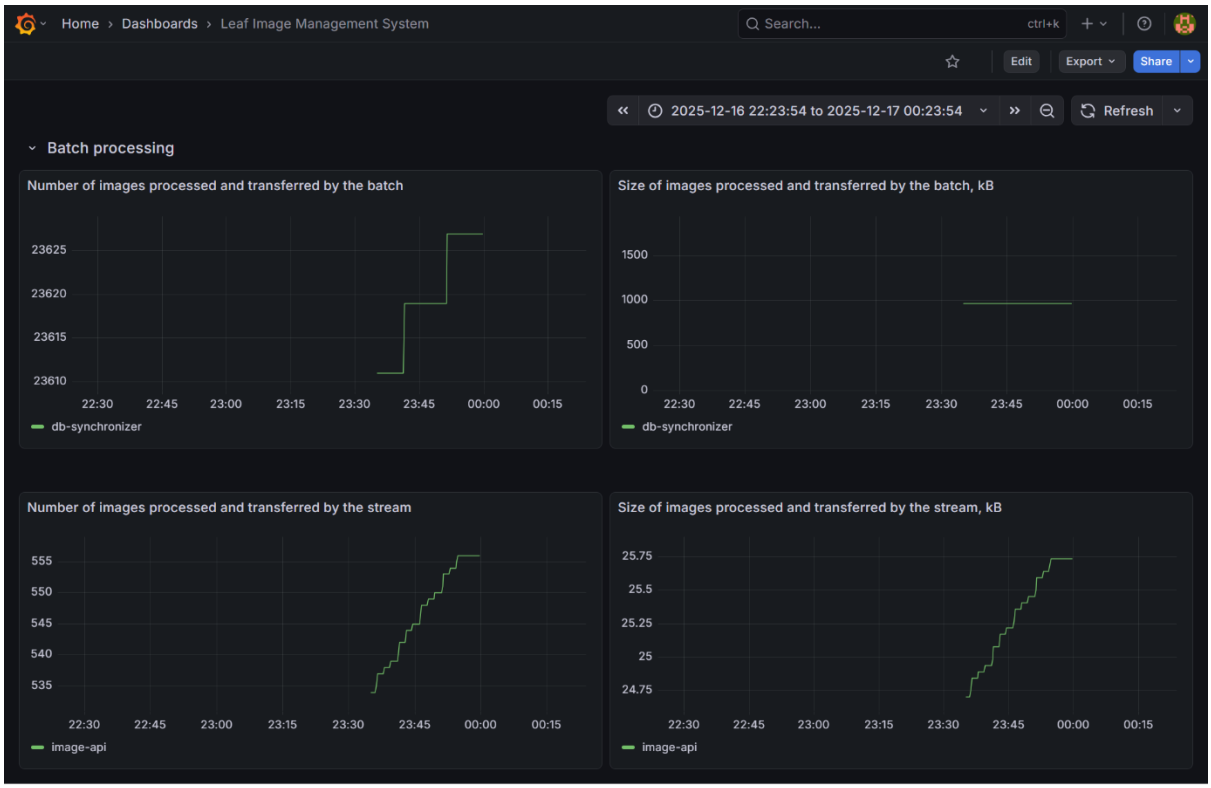

Figure 1 shows ingress

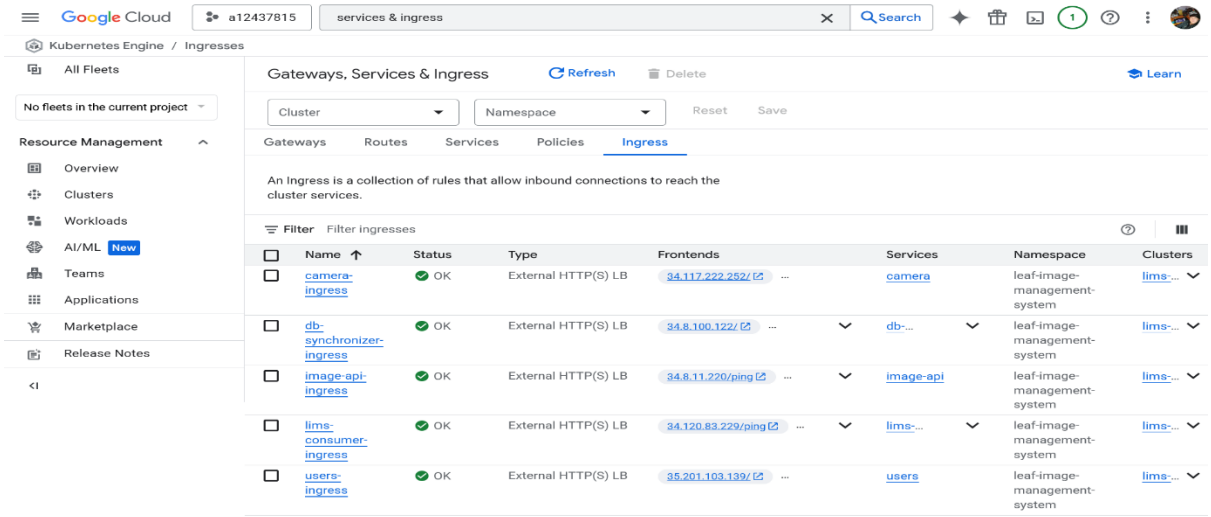

Figure 1 shows the services

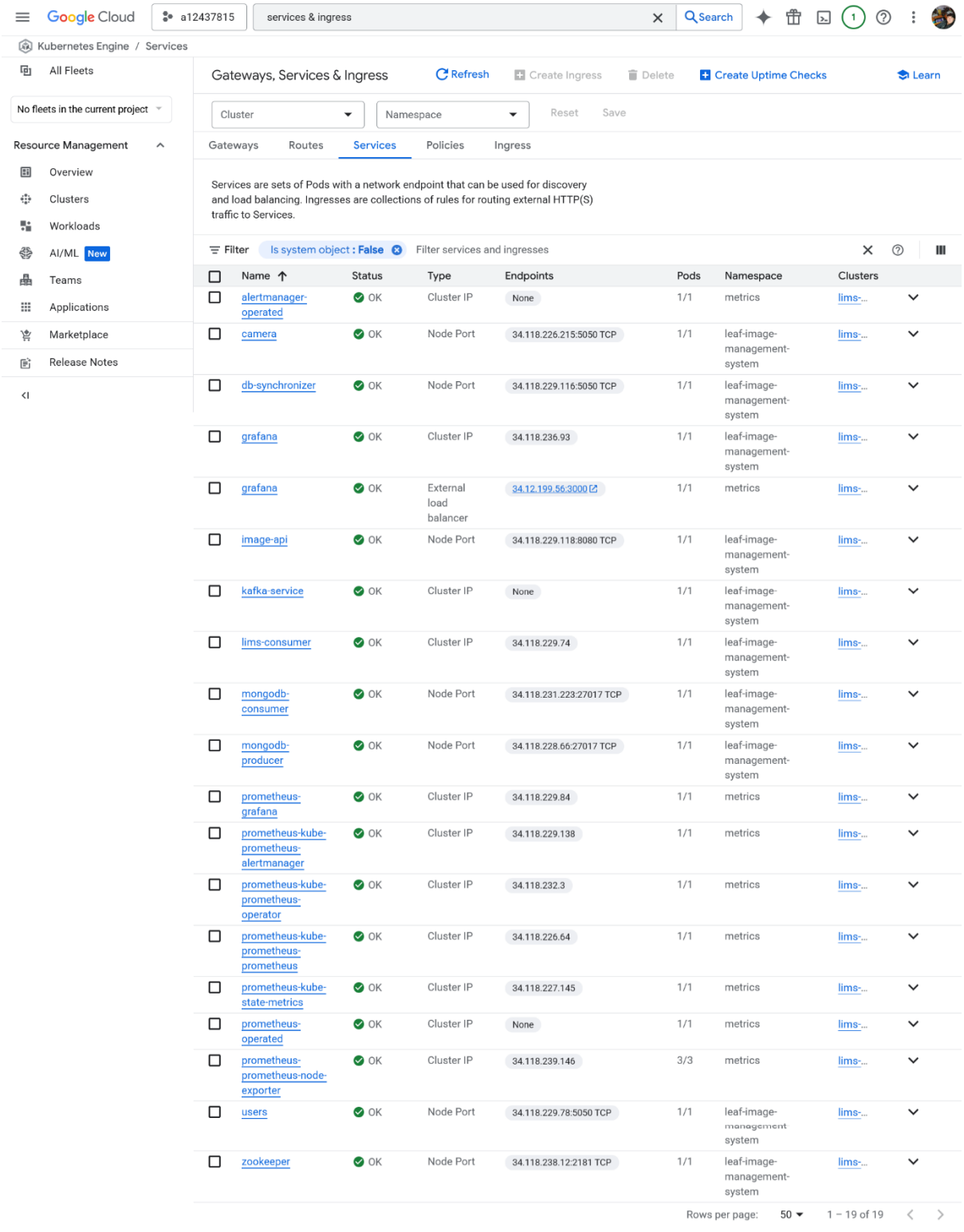

Figure 1 shows workloads