Kafka Consumer

Implementation details, covering setup, your Kafka-based consumption service, and other pertinent aspects

There are several reasons for having the decoupled DB (producer-consumer) in the first place:

1. Performance. If a bunch of consumers query the producer DB, it can stress the database, slowing down the producer service. Each consumer would independently hit the producer DB for the same data, multiplying the load.
2. Coupling. Any schema change, maintenance, or downtime in the producer DB directly impacts all consumers.
3. Replay. Once the consumer reads the data, if it misses it, there's no built-in way to "replay" the event without Kafka
4. Transform. Often, data needs to be extracted, transformed, and only the relevant attributes must be loaded and used by consumers.

The main purpose of the consumer to be developed is to "consume" the images from the producer DB to the consumer DB, ensuring the data between the databases is synchronized. This shall happen in real time, so as a background process or daemon.

Consumer Job

To achieve that, I made use of the kafka-python and pymongo libraries. Consumer Job first creates a connection to the Mongo database and the Kafka bootstrap servers:

```python
# Lifespan
def main():
    # Mongo
    logger.info("Connecting to Mongo")
    db = get_db()

    # Kafka
    logger.info("Connecting to Kafka")
    consumer = KafkaConsumer(
        *TOPICS,
        bootstrap_servers=KAFKA,
        group_id=GROUP_ID,
        auto_offset_reset="earliest",  # when there's no committed offset yet
        enable_auto_commit=False,
        value_deserializer=lambda x: json.loads(x.decode("utf-8")),
    )

    logger.info("Kafka consumer started")
    consume_messages(consumer, db)
```

I define the topics I want to subscribe to, the consumer group ID, and disable auto-committing. Importantly, I set `auto_offset_reset` to earliest to start reading from the beginning of the

topic if there is no committed offset yet, and deserialize the received payload to a Python dictionary object for easier access later.
Then it blocks in `for m in consumer` waiting for Kafka messages to arrive. If no messages are available, the broker waits up to `fetch_max_wait_ms` (default 500 ms) to see if new messages arrive. For each received message, it upserts the value in the consumer database, ensuring it stays up-to-date with the producer database. And since we have soft delete (changing the flag), this is enough. Consumer Job also logs the changes it made.

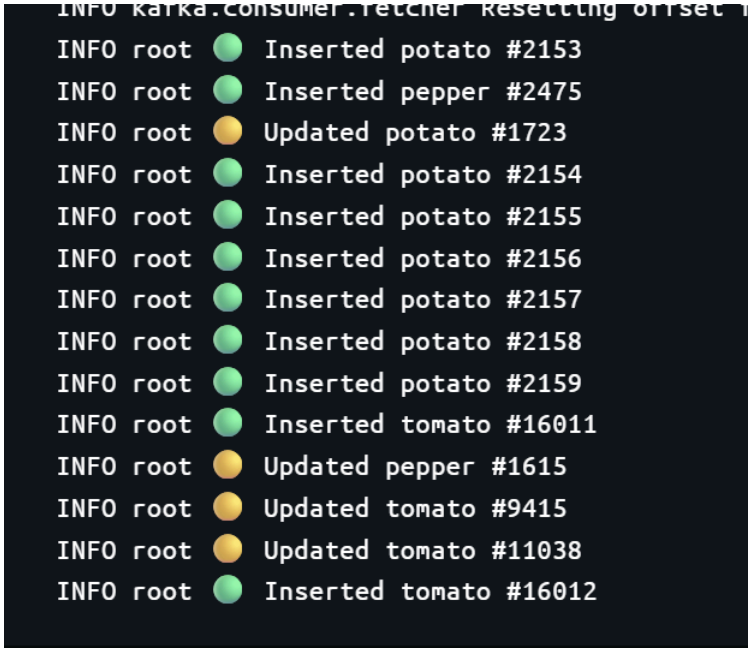
```python
# Consumer loop
def consume_messages(consumer: KafkaConsumer, db: Database):
    for m in consumer:
        try:
            value, topic = m.value, m.topic

            collection = get_collection_by_type(db, topic)

            result = collection.update_one(
                {"id": value["id"]}, {"$set": value}, upsert=True
            )

            if result.upserted_id:
                logger.info(f"🟢 Inserted {topic} #{value['id']}")
            elif result.modified_count > 0:
                logger.info(f"🟡 Updated {topic} #{value['id']}")
            else:
                logger.info(f"🔴 No changes {topic} #{value['id']}")
        except Exception as e:
            logger.error(f"Error processing message: {e}")
        finally:
            consumer.commit()

```



Error handling

During looping

In case an error is encountered parsing the message to value and topic, or e.g., connection to the database is dropped, the message is dropped for simplicity (with `finally: commit()`).

One solution to this would be to move the commitment to the successful execution only (`try` part) - this way, Consumer Job will refetch the message from the last offset and retry until success. But if the problem is in the message itself, it becomes a dead letter, stopping the synchronisation job indefinitely. A compromise would be to set the retry backoff to N times or send failed messages to a separate dead-letter topic.

I tried killing the Kafka broker while looping for messages as well. Kafka consumer enters reconnection loop without throwing an exception and exiting until it can successfully reach the broker.

During creation

I deliberately decided not to introduce /ready and /live probes to the job and not to silence the critical errors. So if any error is thrown in `main()`, the application with exit with an error and Kubernetes will restart it.

Sync vs Async

Consumer shall also expose two endpoints:

`/image-plant/{image_type}/{image_id}` and `/image-plant/{image_type}/total`. First, I put it all in one FastAPI app, making the consumer job run as a background process, and the foreground was serving api requests. This required connections to Kafka and the MongoDB, made asynchronous to yield control from the job to the api when waiting for new messages to arrive or making changes to the database. But shortly after, I realized that this is bad design in the first place to have API and Job in a single app, so I decided to decouple them into two microservices, namely Consumer Api and Consumer Job. This also removed the need to make Kafka and MongoDB async because in Consumer Job thread now has nothing else to do, so there is no need to yield control to anything.

Consumer API

Consumer API is a simple FastAPI app implementing the required endpoints, but also introducing /ready and /live probes, which are essential for Kubernetes to understand whether the API is not stuck.

```python
# Probes (will hang in case something bad happened)
@app.get("/ready")
async def readiness():
    return "ready"


@app.get("/live")
async def liveness():
    return "alive"


# endpoints
@app.get("/image-plant/{image_type}/total")
def get_total_images(image_type: ImageType):
    collection = get_collection_by_type(db, image_type.value)

    total = collection.count_documents({})
    return total


@app.get("/image-plant/{image_type}/{image_id}")
def get_image_by_id(
    image_type: ImageType,
    image_id: int,
):
    collection = get_collection_by_type(db, image_type.value)

    doc = collection.find_one({"id": image_id})
    if not doc:
        raise HTTPException(status_code=404, detail="Image not found")

    doc.pop("_id", None)
    return doc

```

It also defines a global client for interaction with the consumer database.

```python
# Global DB client
db: Database = None


@asynccontextmanager
async def lifespan(app: FastAPI):
    global db

    client = get_client()
    db = get_db(client)

    yield

    client.close()


app = FastAPI(lifespan=lifespan)

```

Sync vs Async

FastAPI allows for async endpoints. However, since Job uses a synchronous MongoDB client, rather than rewriting everything to the async Motor library, I decided to keep the common MongoDB client connection code synchronous and maintain synchronous endpoints. This way, Uvicorn threadpool handles parallel requests instead of async (green) threads. From the docs: PyMongo is thread-safe and provides built-in connection pooling for threaded applications. So, PyMongo handles concurrent requests via connection pooling, so multiple threads can query the DB in parallel.

Challenges

Main challenges here were to come up with this decoupled architecture, decide sync/async, and find the config names for Kafka topics, DB and collection names, and the format of the messages.

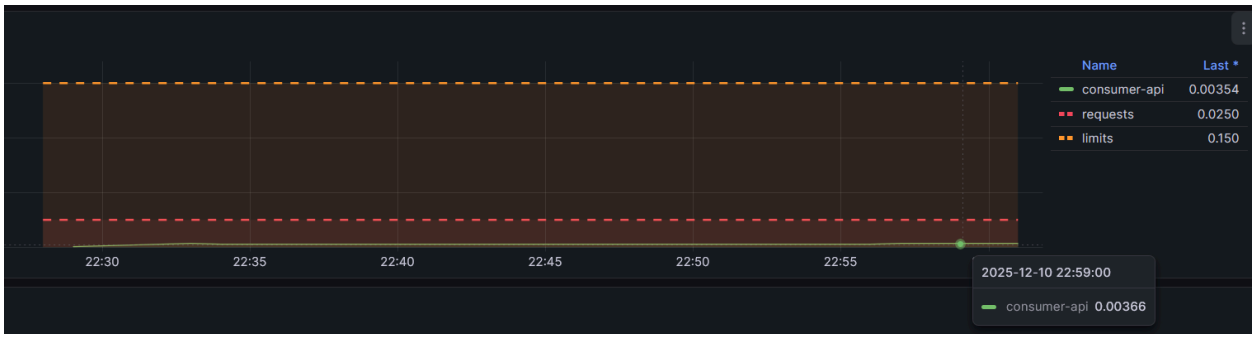
Scalability

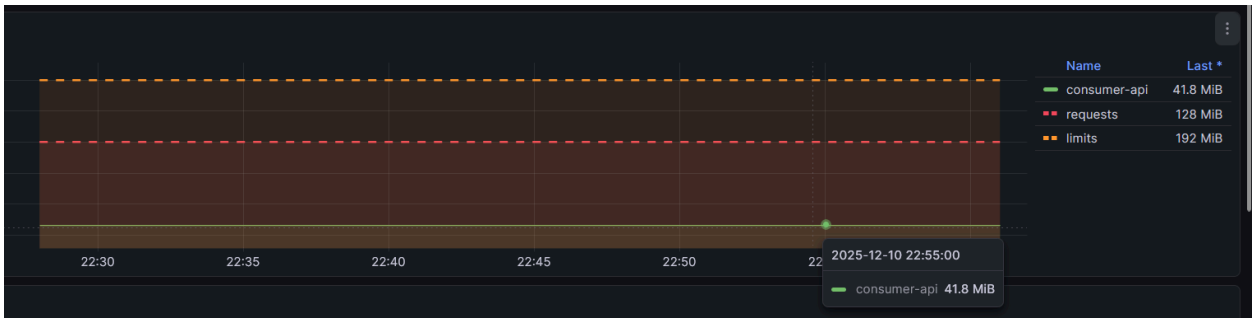
Both the Consumer API and the Consumer Job services have the resource requirements and limits defined, which allow them to burst in case of traffic increase.

API

If the intensity of data flow increased, thead pools may still not be enough, and fully async API routes and mongo connections will still make a difference.

Resource requests and limits I started with:





This clearly showed that both limits and requests must have been decreased, and I decreased them to:

```yaml
resources:
  requests:
    cpu: "10m"
    memory: "64Mi"
  limits:
    cpu: "50m"
    memory: "128Mi"

```

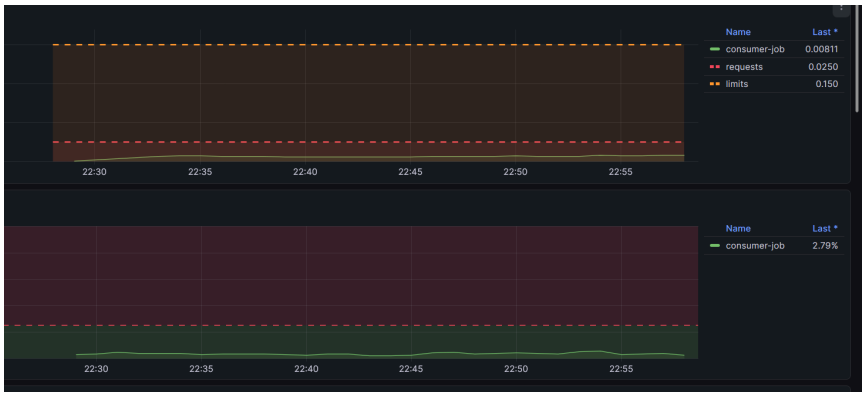
Job

Since the job processes messages one by one, there is not much I could change here. Async will not help here.

What could be done is to introduce Consumer Job replicas, each subscribed to one of the potato/tomato/pepper topics.
Or even more, topics could be divided into partitions, but ensuring images with specific ID always get to the same partition, and Consumer Job replicas will subscribe to those partitions. This way, there will be no race conditions between changes to the same image (because Kafka guarantees the order of messages within a single partition).

To make the number of Consumer Job replicas adjust to the load (scale up and down like HPA), KEDA (Kubernetes Event-Driven Autoscaler) could be introduced, and a ScaledObject for Kafka consumer lag can be created. The advantage is that KEDA automatically scales the number of Kafka consumer pods up and down based on real Kafka lag. The disadvantage is that it has to be learned and configured.

Initial resource requests and limits turned out to be high as well:



So I decreased them to:

```yaml
resources:
  requests:
    cpu: "10m"
    memory: "32Mi"
  limits:
    cpu: "50m"
    memory: "64Mi"
```

On GKE, I dedicate enough resources to the nodes so that Kubernetes pods can scale vertically to their limits in resources in case of CPU or memory load increase.

Batch processing vs Stream processing

The first thing that can be noticed on the Grafana dashboard is that there is a bunch of images already present in the Producer DB and being picked up by the db-syncronizer job. While new images are being pushed to Kafka. This results in a significantly higher number of images processed by batching vs streaming.

The second thing is that I would expect the graph of the number of processed images by the batching to look like a step function with large jumps. However, on the first run (when both the

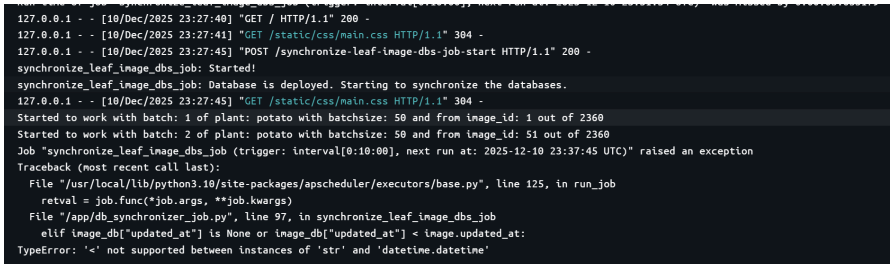
DB synchronizer and the consumer job were up), it rather resembled the performance of stream processing. That could be because of the large number of images to process, and even though it activates once every 10 minutes, it processes a batch of images one by one.

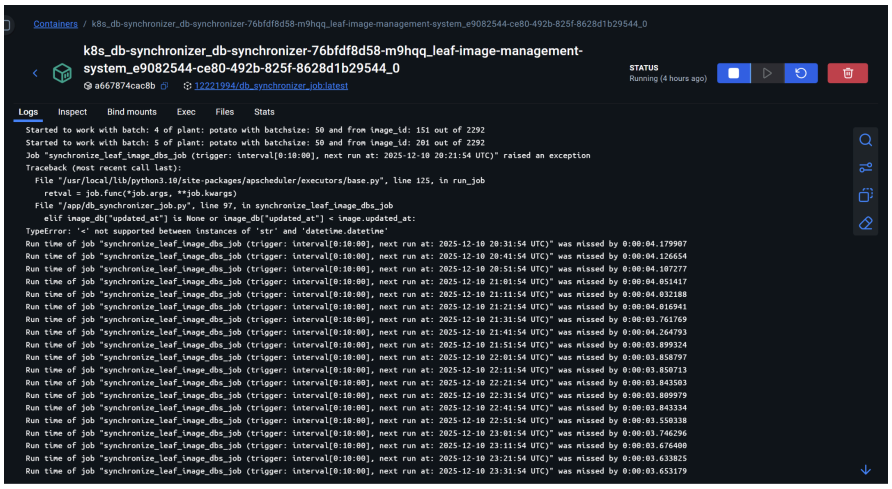
As expected, the number of images processed by the stream grows smoothly because as soon as an image arrives at the Kafka topic, it is picked up and consumed.

A similar observation can be made about the size of the images processed.

After turning the DB synchronizer off, the number of images and the size of the images processed by the batch stopped at the same level, which is expected.

Interestingly, after a restart DB synchronizer tried to process another batch and encountered an error, which stopped the processing at some number of images (without processing the whole batch). As scheduled, it tries again once 10 minutes, but reaches same error.





However, now the Grafana panel showed more like a step function with jumps (of the number of images it had time to process before an error), which confirms previous assumption.

One can improve a batch approach to behave almost like stream processing by using micro-batches, triggering batches as soon as new data arrives (or in short time intervals) instead of on a fixed schedule - in fact, this is what Kafka Consumer stream processing does internally with `fetch_min_bytes` and `fetch_max_wait_ms` parameters. Stream processing is considered better because it provides:

- Real-time, low-latency processing
- Continuous, stable resource usage
- Incremental computation instead of heavy periodic jobs
- Built-in fault tolerance and checkpointing
- Immediate reactions to events (analytics, alerts, automation)

This makes it generally more responsive, efficient, and scalable than traditional batch systems.

Also note that to scale batch processing, one could split data into chunks and fit them into horizontally scaled DB synchronizers, based on a custom consumption lag.

K8s Resources

I introduced Deployment objects for the Consumer API and Consumer Job. Both have resources defined, possibility to change config of the apps with env (like MONGO DB URL or KAFKA BOOTSTRAP SERVERS URL), and specified /ready and /live probes for API.

I also put ClusterIP services in front of the Consumer API and the Consumer Job. And since the API should be made accessible, I also configured Ingress to forward connections to the Consumer API endpoint. So the only entry point to the consumer is through ingress.

Deployment | GKE

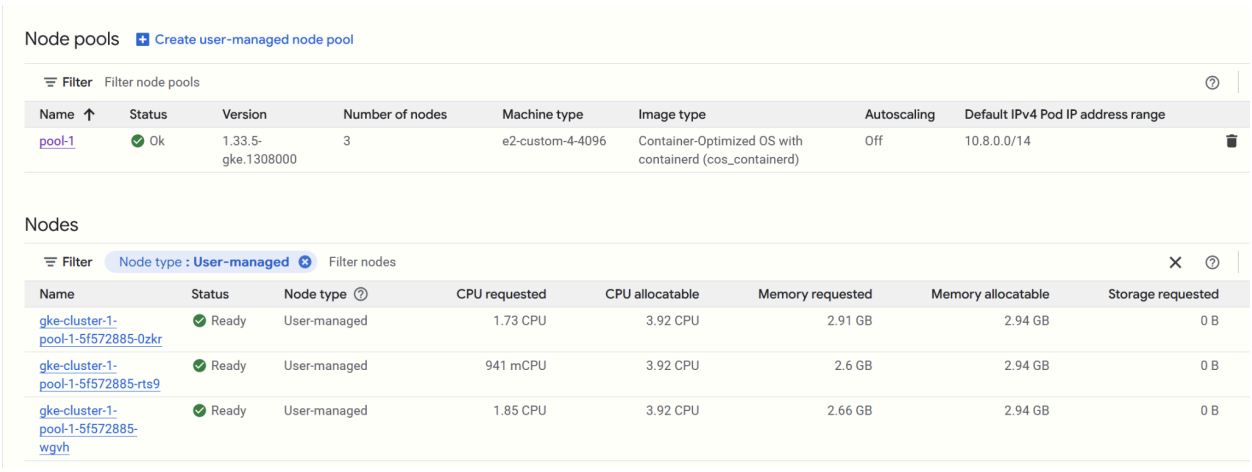
I developed the Consumer locally but tested it on GCP as well.
Moving to GCP, I had to adjust the resource requests/limits (decreasing them), which still works fine locally, so I don't have separate manifests for staging and production. The only difference between staging and production is the presence of an ingress object - it is not necessary for local development since I configured port forwarding to the Consumer API service in open.stg.sh, and if one were to use ingress locally, them will need to start an ingress controller.

As for cluster, I decided to use 3 nodes instead of 1 to make the system more resilient, and this is recommended by GCP.

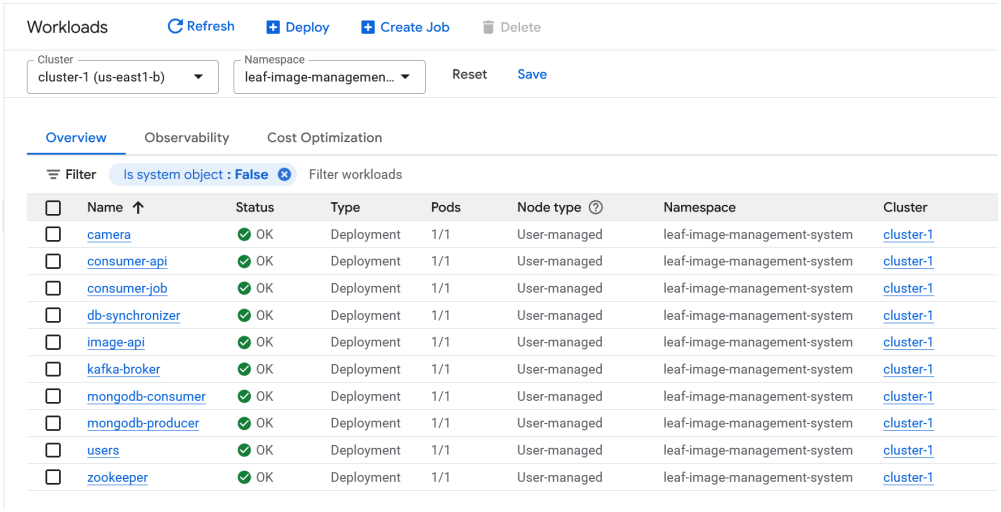
vCPU / Memory

I started with lower values, but successfully deployed with 4 vCPU and 4 GB per node.
An equivalent code snippet of my cluster creation:

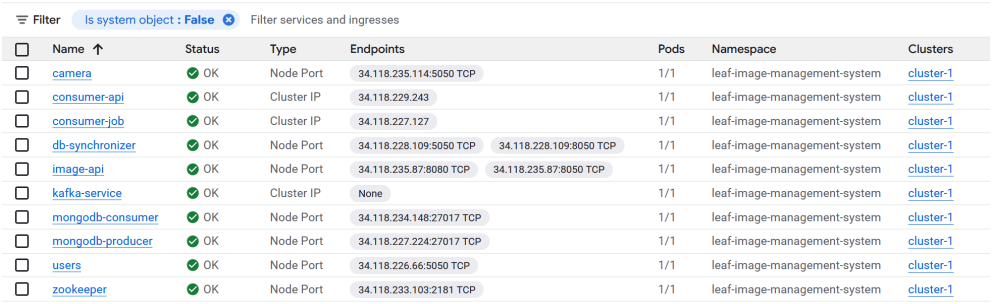
```
gcloud beta container clusters create cluster-1 \
    --project cloud-computing-476715 \
    --zone us-east1-b \
    --no-enable-basic-auth \
    --cluster-version 1.33.5-gke.1308000 \
    --release-channel regular \
    --machine-type e2-custom-4-4096 \
    --image-type COS_CONTAINERD \
    --disk-type pd-standard \
    --disk-size 40 \
```



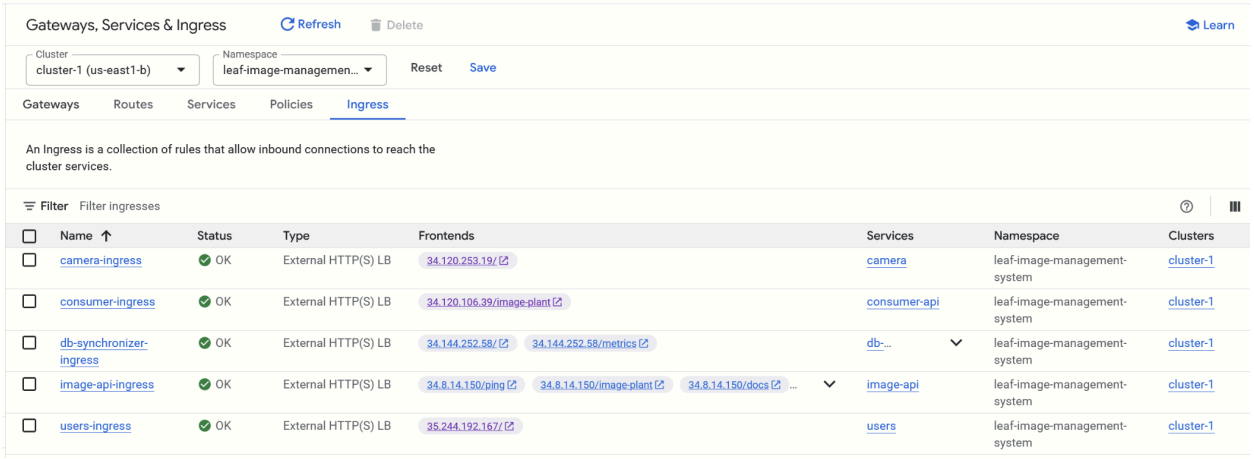
Workloads



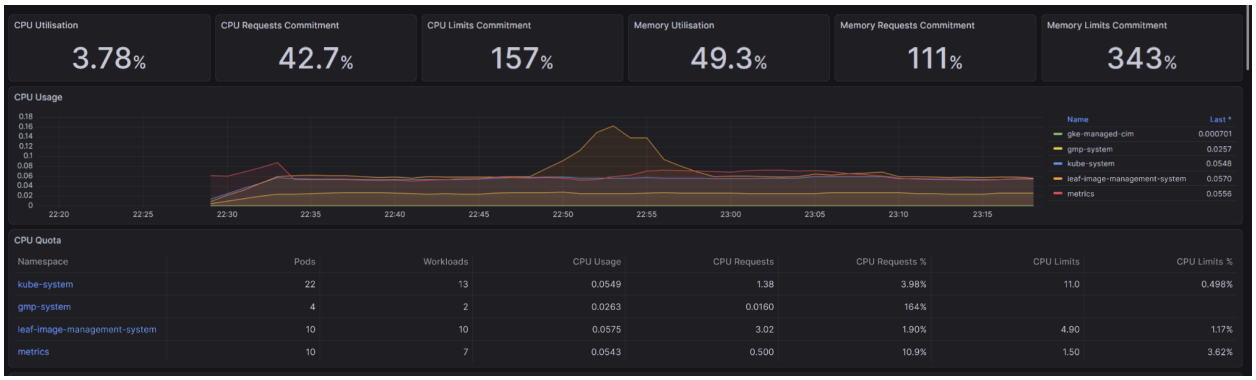
Services



Ingress

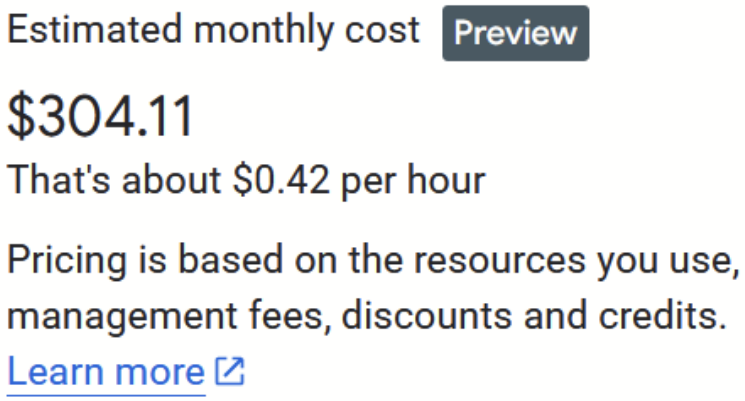


Utilization



Cost estimation

The cluster, as it is, costs around $305/month (with batching and streaming)



On GKE, a batch solution that runs every 10 minutes could be cheaper than a continuously running event-driven architecture if scaling nodes only during batch execution (configure nodes' autoscaling), but if the batches run long enough to occupy the cluster most of the time in between runs, costs approach those of streaming.

For example, a 3-node cluster with `e2-custom-4-4096` nodes as above (~$0.42/hr) would cost roughly $3,680/year if running 24/7, whereas a batch job occupying ~10% of that time could cost ~$368/year, while a streaming setup running continuously would be the full $3,680/year;

It all depends on how “real-time” processing should be and how long it takes to process a batch.

Compared to purchasing own hardware, which might require $5k–10k upfront plus $1k–2k/year in electricity, cooling, and maintenance, GKE provides quite a benefit with auto-scaling, managed upgrades, and higher reliability, making it more flexible and often more cost-effective

for workloads with variable load, though dedicated hardware could be cheaper for consistently high utilization.