Cloud Computing Assignment 2

Scientific Computing Research Group, University of Vienna

1 Motivation

This assignment will help you understand the practical application of event-driven architecture and the advantages of using a stream-based approach for data transmission between services, as compared to classic batch processing.

2 Overview

In this exercise, you will develop a service that interacts with a pre-existing, larger application named the Leaf Image Management System (LIMS) using an event-driven approach. The assignment involves creating a service that continuously and automatically adds photos, modifies their metadata, saves them into the database, and logs these changes to an Apache Kafka cluster. LIMS is written in Python. Your service should be capable of interfacing with the LIMS using a stream-based approach, irrespective of the programming language or framework you choose. To facilitate this interaction, it must connect to the Apache Kafka cluster to consume these photos. Detailed information about the application's operational principles and the requirements for your service implementation can be found below. Additionally, some information about APIs and the context can be found in the *Cloud Computing Assignment 2 (Description)*.

This assignment consists of three tasks:

- Task 1 [55%]: **Develop a service for consuming and saving data from a Kafka cluster. You are required to:**
  - Deploy the existing *LIMS application*, *Kafka cluster*, *Db Synchronizer Job*, and *databases (Producer/Consumer Image DB)* locally using Kubernetes to verify correct data production. Information about the deployment can be found in the README file from the repository.
  - Implement an application that consumes data from at least one topic of the Kafka cluster, ensuring that fault tolerance mechanisms are in place.
  - Connect to an already-established MongoDB database named *Consumer Image DB* to store the consumed data, and implement procedures for creating or updating records based on this data.
  - Implement resource constraints for your application.
  - Deploy your data-consuming system locally using Kubernetes and verify that the deployment works correctly.
  - Test the existing application, the Kafka cluster, and your service locally.

- Task 2 [30%]: **Test and run the existing application, the Kafka cluster, and your service on Google Cloud Platform (GCP) using Google Kubernetes Engine (GKE).**
  - On GKE, the only entry point should be through an Ingress, which forwards requests to your services. The Ingress must include an equivalent of the

    `/image-plant/{image_id}`

    handler to retrieve an image from the consumer database. You can develop this handler as three different handlers, each corresponding to a type of leaf, similar to the handlers used in the Image API (refer to the description file for more details), using REST or gRPC protocols. Alternatively, you could create a single handler for all types of leaves. The decision on how to implement this handler is up to you.
  - Deliver Kubernetes manifest files.
  - Deploy the existing application, the Kafka cluster, and your consumer on GCP.
  - Allocate a budget of up to $10 (always shut down the cluster after testing!).
- Task 3 [15%]: **Compare the bandwidth between the pre-implemented batch processing application and the stream processing implementation you are tasked with.**
  - Connect to Grafana, either locally or on the cloud. It interfaces with the pre-installed Prometheus servers of the LIMS for streaming data, and with the DB Synchronizer Job for batch processing.
  - Import the pre-created dashboard into Grafana, which can be found in the specified section of the repository.
  - Describe the differences in bandwidth between batch and stream processing as observed from the pre-existing Grafana dashboard.

3 Deliverables

The service is located on my GitHub repository. The results of your work should encompass the following points:

- Source code of your service (hosted on our GitLab):
  - Submissions must be pushed to your repository on our GitLab server, in the A2 folder.
  - Create a YAML file for your service that features resource constraints.
  - Your service's Kubernetes manifest YAML files should be included and located in the **/consumer** folder.
- Documentation: Submit a .pdf file on Moodle or a README.md file on GitLab alongside your source code, which should include:
  - Implementation details, covering setup, your Kafka-based consumption service, and other pertinent aspects.

- A description of the challenges faced during your development and deployment.
- An explanation of the opportunities for scalability of the current system and possible bottlenecks, answering the following questions:
  * Is your service scalable? If so, how can you scale your service up or down?
  * What would happen if the intensity of dataflow increased and the number of added or modified photos in the Kafka cluster rose?
  * How can you improve the represented batch-approach to closely approach the efficiency of stream-processing? Why is stream-processing considered better than batch processing?
  * How would the system operate with four Camera Job services in use? Are there alternative methods for configuring the Apache Kafka cluster to transmit this information?
  * How would you describe your approach to scaling your application on GKE?
- A description of the Kubernetes objects (such as deployments and services) and the types of services (NodePort, ClusterIP, or LoadBalancer) you implemented, along with their respective purposes and usage.
- Requirements for deployment: What modifications did you make to enable execution in GKE, if you did not develop directly on GCP?
- Provide the detailed configuration of your cluster on GKE, if it differs from the specification.
- A discussion of the annual costs associated with provisioning the full cluster:
  * In your estimation, which would be more costly on GKE: utilizing a batch solution or adopting an event-driven architecture? Please provide a rough calculation of the annual expenses.
  * How do you believe this compares to the costs of purchasing and maintaining your own hardware?
- Justify the amount of vCPU and memory required by your application.
- Include screenshots showcasing your GKE Cluster, Workloads, Services, and Ingress on GCP.

4 Task Description

High connectivity between services implies a need to transmit and exchange information as rapidly as possible, ideally in (near) real-time. Event-driven architecture is perfectly suited for such scenarios.

An example of this is showcased in our work. In agriculture, data is used to enhance crop quality. For instance, ML-based applications can classify and prevent the spread of various plant diseases by identifying them in photos. In our case, we work with potato, pepper, and tomato leaves. The provided data is sufficient for the task, but if you want to learn more about the data, you can find this information in a separate *Cloud Computing Assignment 2 (Description)* file located in the same directory.

The app, which manages the data flow, automatically generates photos, as farmers can continually take pictures of leaves in the fields. The system automatically adds metadata to these photos and identifies probable foliar diseases. Subsequently, this system sends both the photos and their metadata to a Kafka cluster for consumption by other services. The photo generation process encompasses the following sequential operations:

- Adding new photos, storing them, and encoding them as a byte sequence, along with the associated metadata.
- Identifying potential foliar diseases present in the image using the Leaf Disease Recognizer Job.
- Handling possible user updates to these records, which are automatically made by the Users Job.
- Sending events to Kafka for all the aforementioned addition and modification operations, or using the Db Synchronizer Job to retrieve all data within a certain period of time.

You will need to create a service that connects to the Kafka cluster to retrieve events, which contain information about images and their updates. The entire system can be viewed in Fig. 1.

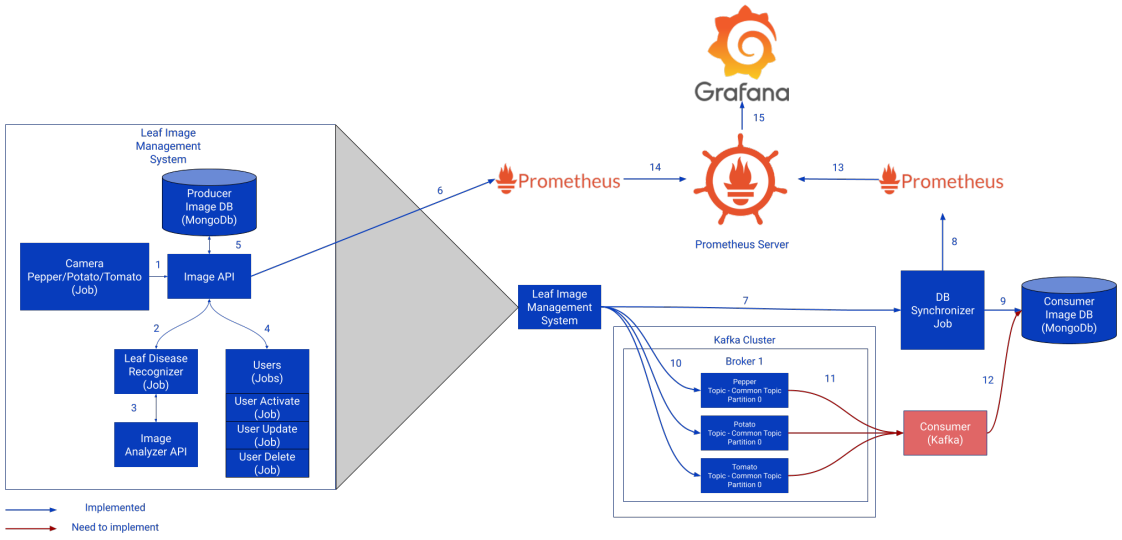

Figure 1: The entire system

The blue part, Prometheus and Grafana have already been implemented. You only need to implement the service colored in red, which is named Consumer (Kafka).

5 Detailed Application Description

5.1 Leaf Image Management System

The Leaf Image Management System (LIMS) allows users to store leaf images and automatically update them, including the capability to recognize leaf diseases. LIMS can be deployed either locally using Kubernetes or on the Google Cloud Platform

(GCP) via Google Kubernetes Engine (GKE). A detailed structure of LIMS is illustrated in Fig. 2.

You do not need to directly use the handlers from APIs of LIMS for your task. However, if you wish to stop the jobs of this service and inspect the image information from the database, some details about the service are required.

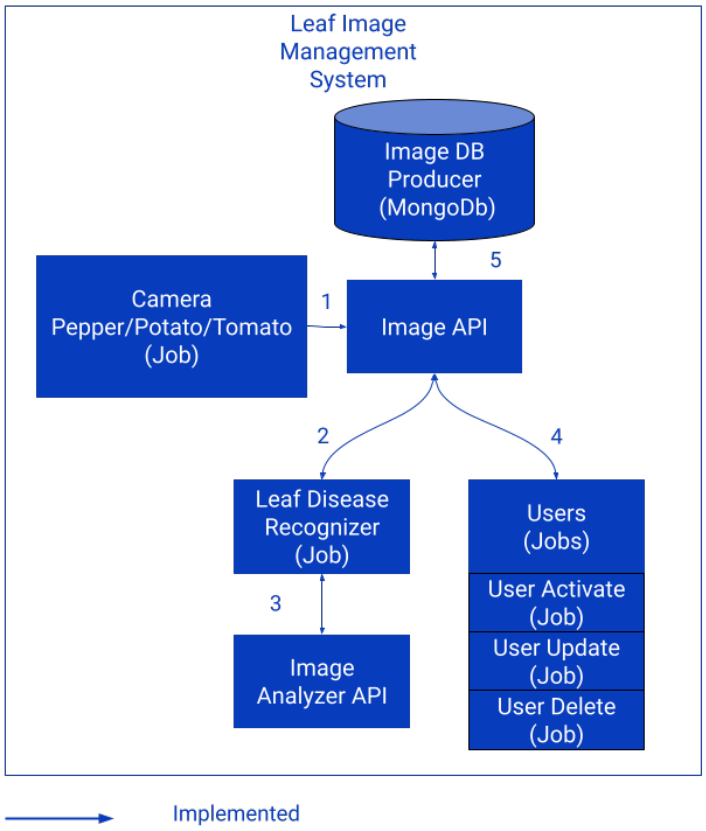

Figure 2: Leaf Image Management System

This schema comprises the following blocks:

Image API. This API performs CRUD operations for images using *Image DB Producer (MongoDB)* and pushes changes to Kafka topics. It transmits all changes (creation, updates, and deletion actions for potato, pepper, and tomato images) to three corresponding topics. For more information about the handlers of this API and about Prometheus, you can refer to the separate *Cloud Computing Assignment 2 (Description)* file.

Image Analyzer API. This API contains ML models that analyze leaf diseases based on different image representations. Similar to the Image API, you can find more information in a separate *Cloud Computing Assignment 2 (Description)* file.

Camera Job. As depicted in Fig. 2, this service handles three distinct types of images. Each instance works with its specific leaf type: pepper, potato, or tomato. This job automatically generates and uploads new photos every 100 seconds.

Leaf Disease Recognizer Job. This job identifies diseases in new images. The Leaf Disease Recognizer inspects photos that have not yet undergone disease recognition and consults the *Image Analyzer API* to determine if the leaf is healthy. Each leaf type possesses its unique set of potential diseases. This process is triggered every 100 seconds.

User Job. This job continuously updates image information and can also activate and delete images using different time intervals for each operation.

Image DB (Producer). This is the producer database and it stores images along with their related information. As depicted in Fig. 1, you can also find the *Image DB (Consumer)*, which contains data after the consuming operation from batch processing using the *DB Synchronizer Job*, which is already implemented, or *Consumer (Kafka)*, which you need to implement according to *Task 1*.

5.1.1 Camera Job (already implemented)

The Camera Job simulates the capture of new leaf photos by farmers to check for various diseases. It randomly generates a type of leaf image (potato, tomato, or pepper) every 100 seconds. Connected to the *Image API*, each generated image is directly sent to the Kafka cluster.

This job features a user interface (UI) (see Fig. 3), located at:

```
{
    http://localhost:5050/ - locally
    camera-job.leaf-image-management-system.svc.cluster
    .local:5050 - kubernetes
}
```

Figure 3: Camera Job - UI

The UI for each job provides several options:

1. Start - Initiates the job. You can commence this job immediately after pressing this button.
2. Pause - Temporarily halts the job's operation and allows for resumption from the paused point.
3. Stop - Ceases the job and restarts from the beginning upon reactivation.
4. Open Logs - Accesses logs to view the results of the job's operations.

You can pause or stop each job to test your Kafka consumer. Additionally, each job has logs, which are represented in Fig. 4.

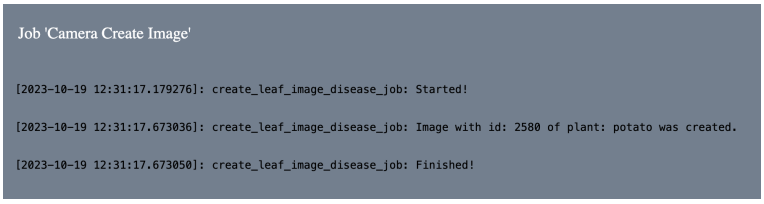

Figure 4: Camera Job - UI (logs)

5.1.2 Leaf Disease Recognizer Job (already implemented)

The Leaf Disease Recognizer Job utilizes the *Image Analyzer API* to classify foliar diseases, and then sends this information to the *Image API* to store the disease classifications for images in the Image DB Producer. It operates every 100 seconds and offers the same functionalities as the Camera Job for Start/Pause/Stop/Open Logs.

This job features a UI (see Fig. 5), located at:

```
{
    http://localhost:5051/ - locally
    leaf-disease-recognizer-job.leaf-image-management-system
    .svc.cluster.local:5050 - kubernetes
}
```

Figure 5: Leaf Disease Recognizer Job - UI

5.1.3 User Job (already implemented)

This job features a UI (see Fig. 6), located at:

```
{
    http://localhost:5052/ - locally
    user-job.leaf-image-management-system.svc.cluster
    .local:5050 - kubernetes
}
```

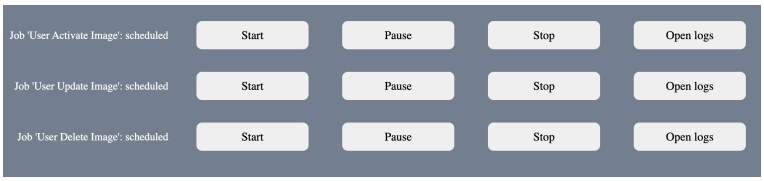

Figure 6: User Job - UI

This service comprises three distinct jobs:

1. Job to activate or deactivate images: This job randomly activates deactivated images and deactivates activated images, simulating user activity. It runs every 300 seconds.
2. Job to update image metadata: This job modifies metadata, including the GPS coordinates and camera ID. It runs every 300 seconds.
3. Job to delete images: This job employs a soft delete approach, where instead of completely deleting data, the system uses a special flag. This job deletes one random image every 700 seconds.

These jobs are also connected to *Image API* and offer the same functionalities for Start/Pause/Stop/Open Logs as the *Camera Job* and *Leaf Disease Recognizer Job*.

5.2 Image DB Producer/Consumer – MongoDB (already implemented)

We use MongoDB to store data, and it is deployed and populated automatically alongside other services.

For the producer, MongoDB contains the following database:

```
imagedbplantproducer
```

For the consumer, MongoDB contains the following database:

```
imagedbplantconsumer
```

Each database has the following collections according to each type of image:

```
imagecolplantpotato
imagecolplanttomato
imagecolplantpepper
```

Each collection consists of the following records. For example:


```
{
  "_id": {
    "$oid": "6522cda0ad2ee195f8bf435a"
  },
  "id": 1,
  "camera_id": 1,
  "data": {
    "$binary": {
      "base64": "/9j/4AAQSkZJRgABAQAAAQABAAD...",
      "subType": "00"
    }
  },
  "gps_coordinates": [
    48.585179,
    16.947841
  ],
  "disease": null,
  "percentage": null,
  "created_at": {
    "$date": "2023-10-08T15:41:12.347Z"
  },
  "updated_at": null,
  "is_active": false,
  "is_deleted": false
}
```


Both the producer and consumer databases are populated automatically. Subsequently, the Db Synchronizer Job runs to synchronize the data between the databases, ensuring they have consistent creation and update data.

5.3 Db Synchronizer Job (already implemented)

There is a service that synchronizes the Image DB Producer and Image DB Consumer databases using a batch approach (see Fig. 7).

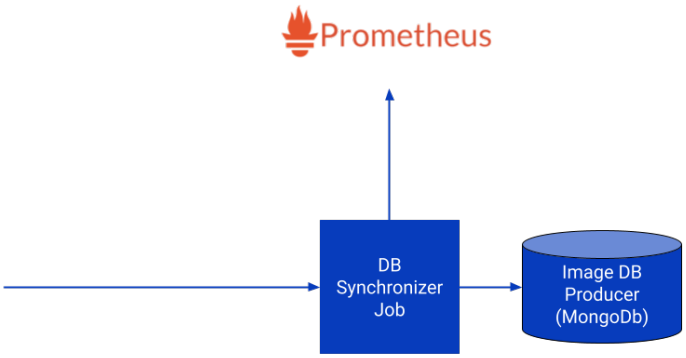

Figure 7: Db Synchronizer Job - Schema

This job has a UI (see Fig. 8) located at:

```
{
    http://localhost:5053/ - locally
    db-synchronizer-job.leaf-image-management-system.svc
    .cluster.local:5050 - kubernetes
}
```

Figure 8: Db Synchronizer Job - UI

The job runs directly after deployment once to ensure that the databases for both producer and consumer are identical and contain the same data. This job runs every 600 seconds and offers the same Start/Pause/Stop/Open Logs options as other jobs.

When you implement your Kafka consumer, you should stop this job, as the synchronization of the databases will be handled by stream processing.

5.4 Kafka Cluster (already implemented)

The Kafka cluster is deployed alongside the LIMS service and contains one broker. Each type of image has its own topic (see Fig. 9).

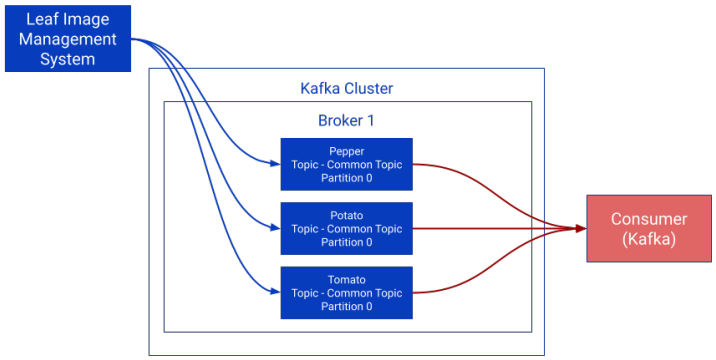

Figure 9: Kafka cluster - Schema

It is recommended to use Conductor (refer to Fig. 10) to display the topics and monitor production.

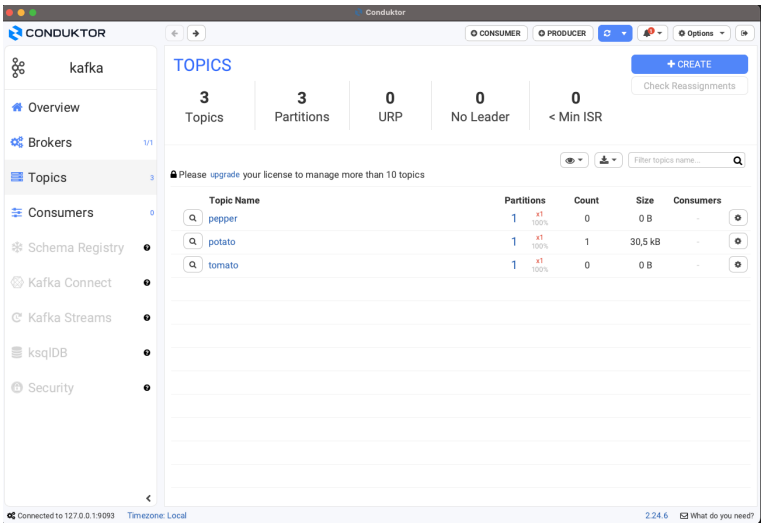

Figure 10: Kafka cluster - Conductor

To access the Kafka cluster, we need to connect to the following ports:

```
{
    http://localhost:9093 - locally
    kafka-service.leaf-image-management-system.svc
    .cluster.local:9093 - kubernetes
}
```

5.5 Prometheus and Grafana (already implemented)

After you have implemented your consumer, to address Task 3, you will need to use Grafana to compare the bandwidth between the pre-implemented batch-processing application using *Db Synchronizer Job* and the stream-processing application using *Kafka Cluster*.

The Prometheus and Grafana cluster is deployed separately, and details about it can be found in the README file located in the project repository. We use a new namespace named 'metrics' to deploy them. Also, you can find more information

about the Prometheus Cluster and metrics for *Image API* and *Db Synchronizer Job* in the *Cloud Computing Assignment 2 (Description)*.

Prometheus collects metrics from the services. You need to utilize Grafana to visualize these metrics in a more convenient way. Grafana can be accessed through the following ports:

```
{
    http://localhost:3000 - locally
    grafana.metrics.svc.cluster.local:3000 - kubernetes
}
```

Grafana is deployed along with the Prometheus and has its own UI (Fig. 11).

Figure 11: Grafana - UI

To compare batch-processing based on *Db Synchronizer Job* and stream-processing based on the Kafka cluster, you need to import an already created dashboard. This information can be found in the README file in the project repository.

5.6 Consumer (to be implemented)

As you can see, the current schema operates using batch processing from the *Db Synchronizer Job*, where each start of this job allows for the transmission of new data from the Image DB Consumer to the Image DB Producer. This approach requires a high level of bandwidth for synchronization, and you could use a stream approach as an alternative.

This service is called the Consumer, and it needs to be implemented for this assignment (see Fig. 12). The main task is to consume at least one topic from the Kafka cluster and create, update, or delete information in the Image DB Consumer to synchronize data between the two databases.

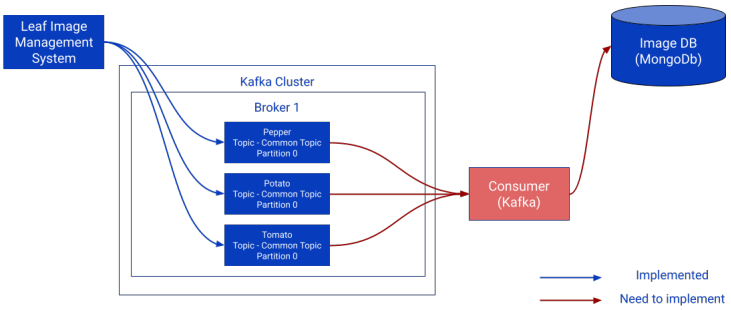

Figure 12: Consumer - Schema

After creating this service, you need to add two .yaml files to deploy the service: one for local deployment and another for cloud deployment. You should use 'env' labels to facilitate environment specification within a single application.

Additionally, as per Task 2, you are required to add a handler:

```
[GET] /image-plant/any_type_of_the_plant/{image_id}
```

This handler corresponds to a handler in the Image API. You can find more information about this handler and the related database data in *Cloud Computing Assignment 2 (Description)*.

When you have deployed the service alongside the entire system, you must conduct a bandwidth comparison experiment and compare the bandwidth of the *Db Synchronizer Job*, which runs every 600 seconds, to the stream-based solution. For this purpose, you can utilize the pre-installed Grafana dashboard to observe the differences.

6 Infrastructure

Every service in this schema, except for Prometheus and Grafana (which use the 'metrics' namespace), operates within a network named 'leaf-image-management-system'. You need to adhere to this ('leaf-image-management-system') schema for the consumer. Also, you must access or download the already-implemented services. The instructions and additional resources can be found at GitHub.

The system includes the following images from DockerHub:

```
12221994/image_api - FastAPI-based API with Prometheus

12221994/image_analyzer_api - FastAPI-based API with
pre-installed ML models for foliar disease classification
using TensorFlow

12221994/users_job - Three Flask-based jobs

12221994/camera_job - Flask-based job

12221994/leaf_disease_recognizer_job - Flask-based job

12221994/db_synchronizer_job - Flask-based job with
```

```
Prometheus

12221994/producer_plant_db – MongoDB database with
pre-installed data for the producer service

12221994/consumer_plant_db – MongoDB database with
pre-installed data for the consumer service

wurstmeister/zookeeper – Zookeeper for the Kafka cluster

wurstmeister/kafka – Kafka cluster
```

To deploy or delete the entire system locally, you can use the following bash scripts located in the repository linked above:

```
start.stg.sh
stop.stg.sh
```

You can always deploy everything manually on your own. The repository contains .yaml files that can be deployed using the 'kubectl' command, as described in the start.stg.sh script. All services have a single instance.

For Prometheus and Grafana, you need to use the following scripts:

```
start-metrics.stg.sh
stop-metrics.stg.sh
```

The README file provides instructions on how to start these services. Remember, you will be working with containerized services.

To deploy these services locally, you need to use your local Kubernetes cluster. This can be set up with Minikube or through the Kubernetes Extensions of Docker Desktop.

To deploy or delete the entire system on GCP, you can use the following bash scripts located in the above repository:

```
start.prod.sh your_project_id (from GCP)
stop.prod.sh
```

Additionally, to use Prometheus and Grafana on GCP you need to use the following scripts:

```
start-metrics.prod.sh
stop-metrics.prod.sh
```

Remember that you can always stop all jobs in the service and prevent the creation or update of images.

It is preferable to deploy metrics locally to avoid wasting credits, as you can still observe the differences between batch and stream processing using metrics even in a local environment.

Some services might be slower than others, especially the deployment of databases, because time is needed to create the collections and fill these databases with information about images. Afterward, additional time is required to synchronize these databases using the Db Synchronizer Job.

7 Resources

These resources will help you understand the question more deeply, as well as prepare your infrastructure for the services:

- Conductor – https://www.conduktor.io/get-started/
- Docker – https://docs.docker.com/
- FastAPI – https://fastapi.tiangolo.com/
- FlaskAPI – https://flask.palletsprojects.com/en/3.0.x/
- Grafana (tool to show metrics) – https://grafana.com/docs/grafana/latest/setup-grafana/installation/kubernetes/
- Kafka – https://kafka.apache.org/documentation/
- Confluent-Kafka (library to use Kafka with Python) – https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html
- Kubernetes – https://kubernetes.io/docs/home/
- MongoDb – https://www.mongodb.com/docs/
- Minikube (to deploy Kubernetes locally) – https://minikube.sigs.k8s.io/docs/start/
- Prometheus (tool to preserve metrics) – https://grafana.com/docs/grafana-cloud/monitor-infrastructure/kubernetes-monitoring/configuration/configure-infrastructure-manually/prometheus/prometheus-operator/
- PyMongo (library to use MongoDB with Python) – https://pymongo.readthedocs.io/en/stable/

Additionally, there are some articles that might be useful for creating your own Kafka consumer:

- Kafka And Python – https://medium.com/fintechexplained/kafka-and-python-
- What is Kafka Server and how to use it in C-Sharp – https://medium.com/@i.bzdgn/what-is-kafka-server-and-how-to-use-it-in-c-3244cf99c81
- Kafka Using Java. Part 1. – https://medium.com/pharos-production/kafka-using-java-e10bfeec8638
- Kafka Using Java. Part 2. – https://medium.com/pharos-production/kafka-using-java-part-2-83fd604ed627