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, and logs these changes to an Apache Kafka cluster. LIMS is written in Python. Your service should be capable of interfacing with the LIMS, 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.

This assignment consists of three tasks:

- Task 1 [55%]: **Develop a service for consuming and logging data from a Kafka cluster. You are required to:**
  - Deploy the existing LIMS application, Kafka cluster, Db Synchronizer Job, and databases locally using Kubernetes to verify correct data production.
  - 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 to preserve the consumed data and implement the creation or updating of records based on the consumed data.
  - Implement resource constraints for your application.
  - Deploy your data-consuming system locally using Kubernetes and verify that the process 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 should forward requests to your services.
  - Deliver Kubernetes manifest files.
  - Deploy the existing application and the Kafka cluster 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 you are tasked to implement:**
  - Connect to Grafana, which is based on the pre-installed Prometheus servers of the LIMS, for streaming data, and to the DB Synchronizer Job for batch processing.
  - Describe the differences in bandwidth between batch and stream processing as observed from the pre-existing Grafana dashboard.

3 Deliverables

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 **/manifests** 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 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 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?
- Detailed configuration of your cluster on GKE (if different 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 of 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 app, which manages the data flow for you, 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, preserving 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 get 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.

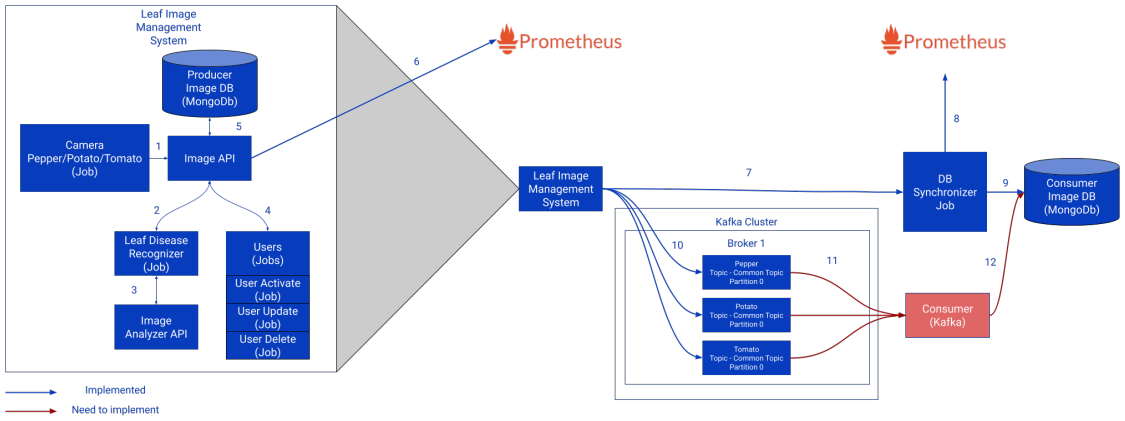

Figure 1: The entire system

The blue part and Prometheus have already been implemented. You only need to implement the service colored in red.

5 Detailed Data Description

The ML visual categorization application identifies the categories of foliar diseases for the following leaves:

- Potato (3 types of leaves)
- Pepper (2 types of leaves)
- Tomato (10 types of leaves)

As observed, each type of leaf can have various possible diseases that the system can detect. Fig. 2 illustrates that each type has its own set of diseases, where each disease constitutes a class. For example, potato leaves encompass three classes.

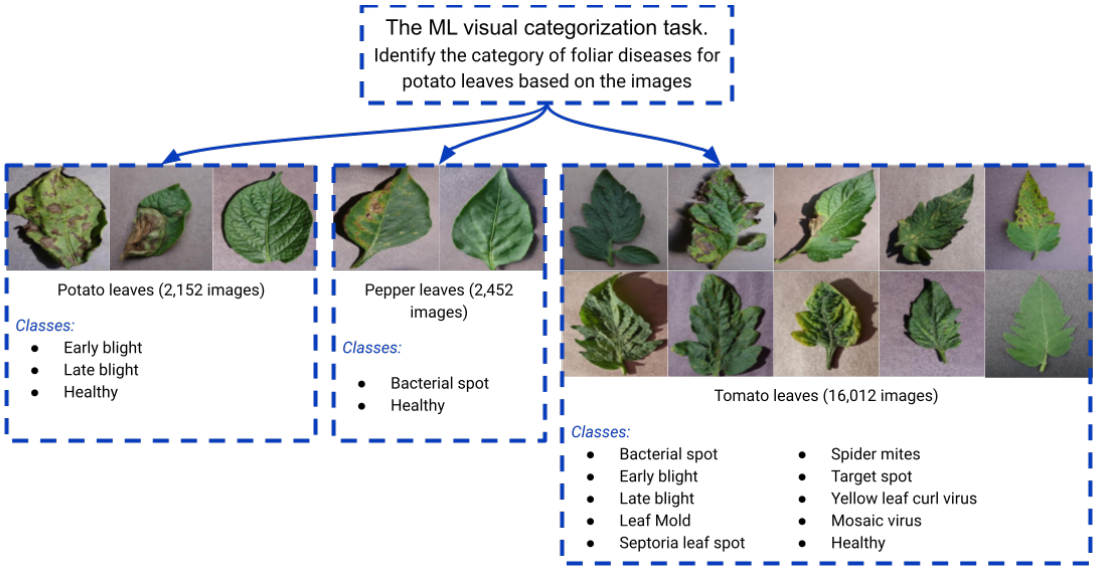

Figure 2: Leaf Diseases Classes

6 Detailed Application Description

6.1 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. 3.

You do not need to directly use the handlers 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.

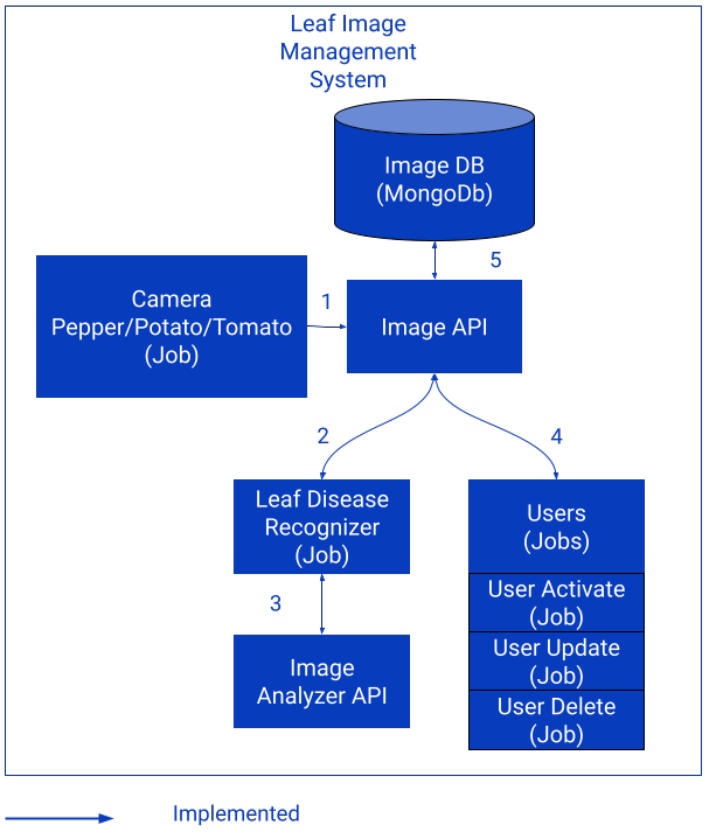

Figure 3: Leaf Image Management System

This schema comprises the following blocks:

Image API. This API performs CRUD operations for images using 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.

Image Analyzer API. This API contains ML models that analyze leaf diseases based on different image representations.

Camera Job. As depicted in Fig. 3, 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 30 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 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 information and can also activate and delete images.

Image DB. This database stores images along with their related information.

6.1.1 Image API (already implemented)

It is essential to delve deeper into the Image API, as it produces messages that you will need to consume.

This service features an API with the following handlers:

1. **Ping the service's status.**

   Request has no attributes. This handler checks the service's operational status.

   ```
   [GET] /ping
   ```

   Response provides the following information if the service is functioning:

   ```
   "Hello, I am alive"
   ```

   This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

2. **Get a potato/tomato/pepper image with metadata by id.**

   Request has a consistent structure for all three handlers and only requires the image's id, where ids are sequential starting from 1:

   ```
   [GET] /image-plant/potato/{image_id}
   [GET] /image-plant/tomato/{image_id}
   [GET] /image-plant/pepper/{image_id}
   ```

   Example:

   ```
   [GET] /image-plant/potato/1
   ```

   Response also maintains a consistent structure for these three handlers (content-type: application/json):

   ```
   {
       "id": 1,
       "camera_id": 1,
       "data": "ffd8ffe000104a46...", (encoded-image-data)
       "gps_coordinates": [
           48.476563,
           16.585654
       ],
       "created_at": "2023-10-08T15:39:49.123123",
   ```

```
    "updated_at": "2023-10-08T15:39:49.123123",
    "is_active": false,
    "is_deleted": false,
    "disease": null,
    "percentage": null
}
```

This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

3. **Get the total count of potato/tomato/pepper images.**

   Request has a consistent structure for all three handlers and does not require any data for transfer:

   ```
   [GET] /image-plant/potato/total/
   [GET] /image-plant/tomato/total/
   [GET] /image-plant/pepper/total/
   ```

   Response maintains a similar structure for all three handlers (content-type: application/json):

   ```
   {
     "total_images": 2157
   }
   ```

   This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

4. **Get a random potato/tomato/pepper image based on filters.**

   Request has a consistent structure for all three handlers and includes two optional flags:

   ```
   is_active
   has_disease
   ```

   These flags are utilized to obtain the appropriate random image. For example, to retrieve only an active random image or a random image that contains information about diseases.

   ```
   [GET] image-plant/potato/random/
   [GET] image-plant/tomato/random/
   [GET] image-plant/pepper/random/
   ```

   Example:

   ```
   image-plant/potato/random/
   image-plant/potato/random/?is_active=true
   image-plant/potato/random/?has_disease=true
   image-plant/potato/random/?is_active=true
   &has_disease=true
   ```

Response maintains a consistent structure for these three handlers (content-type: application/json) and mirrors the model from the first handler:

```
{
    "id": 1,
    "camera_id": 1,
    "data": "ffd8ffe000104a46...", (encoded-image-data)
    "gps_coordinates": [
        48.476563,
        16.585654
    ],
    "created_at": "2023-10-08T15:39:49.123123",
    "updated_at": "2023-10-08T15:39:49.123123",
    "is_active": false,
    "is_deleted": false,
    "disease": null,
    "percentage": null
}
```

This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

5. **Get the batch of potato/tomato/pepper images.**

   Request has a consistent structure for all three handlers and includes two optional flags:

   ```
   is_active
   has_disease
   ```

   These flags, identical to those in the fourth handler, are employed to fetch the appropriate random images. For instance, to retrieve only active random images or random images containing disease information.

   Additionally, there are required parameters:

   ```
   from_id
   limit
   ```

   These parameters allow users to specify a quantity of images starting from a particular number. Using flags in conjunction with these parameters facilitates fetching a specific number of active images from a subsequent list, or images marked with certain diseases.

   ```
   [GET] image-plant/potato
   [GET] image-plant/tomato
   [GET] image-plant/pepper
   ```

   Example:

```
[GET] /image-plant/potato/?from_id=2&limit=2
[GET] /image-plant/potato/?from_id=2&limit=2
&has_disease=false
[GET] /image-plant/potato/?from_id=2&limit=2
&is_active=true
[GET] /image-plant/potato/?from_id=2&limit=2
&is_active=true&has_disease=false
```

Response maintains a consistent structure for these three handlers (content-type: application/json):


```
{
  "image_list": [
    {
      "id": 2,
      "camera_id": 1,
      "data": "ffd8ffe000104a46...", (encoded-image-data)
      "gps_coordinates": [
        48.618306,
        16.87287
      ],
      "created_at": "2023-10-08T15:39:49.668000",
      "updated_at": "2023-10-08T15:39:49.668000",
      "is_active": false,
      "is_deleted": false,
      "disease": null,
      "percentage": null
    },
    {
      "id": 3,
      "camera_id": 1,
      "data": "cf719c28f4c1c1ad...", (encoded-image-data)
      "gps_coordinates": [
        48.038826,
        16.070844
      ],
      "created_at": "2023-10-08T15:40:43.119000",
      "updated_at": "2023-10-08T15:40:43.345000",
      "is_active": false,
      "is_deleted": false,
      "disease": null,
      "percentage": null
    }
  ],
  "total_images": 2157
}
```


This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

6. **Create a potato/tomato/pepper image.**

   Request includes the following parameters:

   ```
   camera_id
   gps_coordinates
   ```

   These parameters apply to all three handlers.

   ```
   [PUT] /image-plant/potato
   [PUT] /image-plant/tomato
   [PUT] /image-plant/pepper
   ```

   The remaining data for images is added automatically.

   Example:

   ```
   {
     "camera_id": 1,
     "gps_coordinates": [
       48.476563,
       16.585654
     ]
   }
   ```

   Response maintains a consistent structure for these three handlers (content-type: application/json) and mirrors the model from the first handler:

   ```
   {
     "id": 2158,
     "camera_id": 1,
     "data": "ffd8ffe000104a4...", (encoded-image-data)
     "gps_coordinates": [
       48.476563,
       16.585654
     ],
     "created_at": "2023-10-09T18:43:37.500688",
     "updated_at": "2023-10-09T18:43:37.500692",
     "is_active": false,
     "is_deleted": false,
     "disease": null,
     "percentage": null
   }
   ```

   This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

7. **Update a potato/tomato/pepper image.**

   Request maintains a consistent structure for all three handlers and includes the following parameters:

```
id
camera_id
gps_coordinates
disease
percentage
is_active
is_deleted
```

These parameters apply to all three handlers.

```
[POST] /image-plant/potato
[POST] /image-plant/tomato
[POST] /image-plant/pepper
```

Example:


```
{
  "id": 1,
  "camera_id": 1,
  "gps_coordinates": [
    48.476563,
    16.585654
  ],
  "disease": "Healthy",
  "percentage": 0.995,
  "is_active": true,
  "is_deleted": false
}
```


Response also maintains a consistent structure for these three handlers and mirrors the model from the first handler (content-type: application/json):


```
{
  "id": 1,
  "camera_id": 1,
  "data": "ffd8ffe000104a4...", (encoded-image-data)
  "gps_coordinates": [
    48.476563,
    16.585654
  ],
  "created_at": "2023-10-08T15:39:49.666000",
  "updated_at": "2023-10-09T19:05:14.792000",
  "is_active": true,
  "is_deleted": false,
  "disease": "Healthy",
  "percentage": 0.995
}
```


This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

8. **Delete a potato/tomato/pepper image by id.**

   Request has a consistent structure for all three handlers and requires only the image's id, with ids being sequential starting from 1:

   ```
   [DELETE] /image-plant/potato/{image_id}
   [DELETE] /image-plant/tomato/{image_id}
   [DELETE] /image-plant/pepper/{image_id}
   ```

   Example:

   ```
   [DELETE] /image-plant/potato/1
   ```

   Response also maintains a consistent structure for these three handlers (content-type: application/json):

   ```
   {
     "message": "Image deleted successfully"
   }
   ```

   This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

Image API features a Swagger interface (see Fig. 4) that can be accessed via:

```
{
http://localhost:8080/docs#/ - locally
image-api.leaf-image-management-system.svc.cluster
.local:8080 - kubernetes
}
```

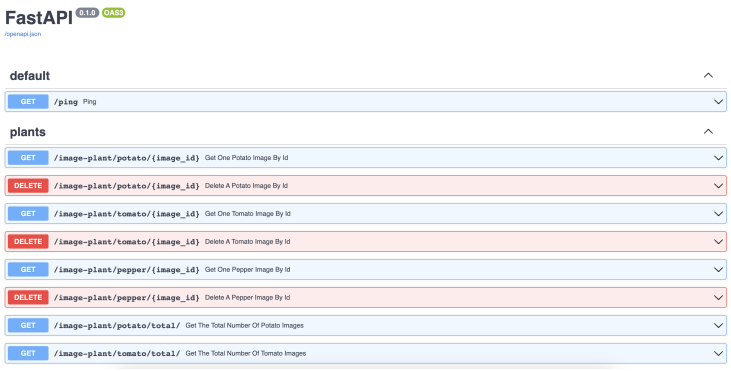

Figure 4: Image API - Swagger

Additionally, Image API provides an open Prometheus port (see Fig. 5):

```
{
http://localhost:8050 - locally
image-api.leaf-image-management-system.svc.cluster
.local:8050 - kubernetes
}
```

Prometheus facilitates the retrieval of service metrics, which are utilized in Grafana. For the Image API, it tracks the number of metadata changes for images, as well as the size of messages sent to the Kafka cluster.

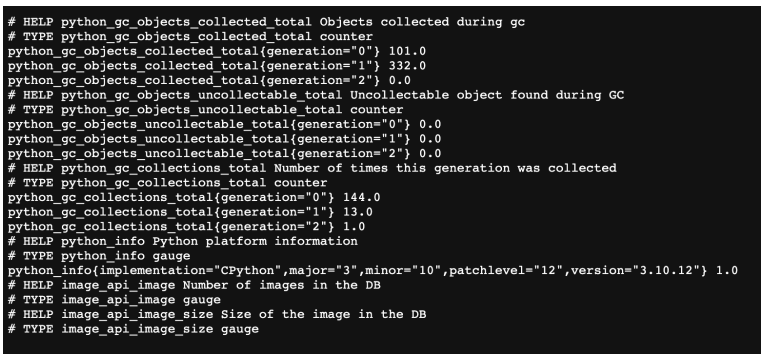

Figure 5: Image API - Prometheus

6.1.2 Image Analyzer API (already implemented)

The Image Analyzer API is used to analyze potential foliar diseases. This service includes an API with the following handlers:

1. **Ping the service's status.**

   Request has no attributes. This handler allows to check if the service works:

   ```
   [GET] /ping
   ```

   Response provides the following information if the service is operational:

   ```
   "Hello, I am alive"
   ```

   This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

2. **Classify diseases of potato/tomato/pepper leaves based on images.**

   Request requires .jpeg images of leaves to be recognized and classified for diseases. The specifics for these handlers are detailed below:

   ```
   [POST] /potato/disease-classify
   [POST] /tomato/disease-classify
   [POST] /pepper/disease-classify
   ```

   Response from these handlers provides the predicted classification of the image and the confidence level. Each handler is designated for each type of leaf:

   ```
   {
     "predicted_class": "Healthy",
     "confidence": 0.8067989945411682
   }
   ```

   This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

3. **Classify potato/tomato/pepper diseases from images (hex representation) decoded from bytes.**

   Request includes the hex representation of images. However, internally, these handlers function identically to those in point 2. Each handler is designated for a specific type of leaf:

```
[POST] /potato/disease-classify-by-bytes
[POST] /tomato/disease-classify-by-bytes
[POST] /pepper/disease-classify-by-bytes
```

   Example:

```
{
  "hex_bytes": "ffd8ffe000104a4649460...."
}
```

   Response from these handlers, similar to the handlers in point 2, provides the predicted classification of the image along with the confidence level.

```
{
  "predicted_class": "Bacterial Spot",
  "confidence": 0.78
}
```

   This response returns a 200 status code if the operation was successful, and a 4xx status code otherwise.

Image Analyzer API has a Swagger (Fig. 6), which can be open by:

```
{
http://localhost:8081/docs#/ - locally
image-analyzer-api.leaf-image-management-system.svc.cluster
.local:8080 - kubernetes
}
```

Figure 6: Image Analyzer API - Swagger

6.1.3 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 30 seconds. Connected to the Image API, each generated image is directly sent to the Kafka cluster.

This job features a UI (see Fig. 7), located at:

```
{
http://localhost:5000/ – locally
camera-job.leaf-image-management-system.svc.cluster
.local:5000 – kubernetes
}
```

Figure 7: 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.

6.1.4 Leaf Disease Recognizer Job (already implemented)

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

```
{
http://localhost:5001/ – locally
leaf-disease-recognizer-job.leaf-image-management-system.svc.cluster
.local:5000 – kubernetes
}
```

Figure 8: Leaf Disease Recognizer Job - UI

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 database). It operates every 100 seconds and offers the same functionalities as the Camera Job for Start/Pause/Stop/Open Logs.

6.1.5 User Job (already implemented)

This job features a UI (see Fig. 9), located at:

```
{
http://localhost:5002/ - locally
user-job.leaf-image-management-system.svc.cluster
.local:5000 - kubernetes
}
```

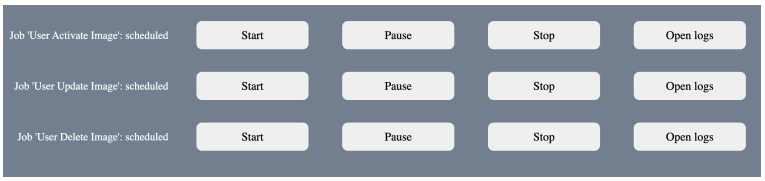

Figure 9: User Job - UI

"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 offer the same functionalities for Start/Pause/Stop/Open Logs as the other jobs from the aforementioned services.

6.2 Mongo Database for producer/consumer (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:

```
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.

6.3 Db Synchronizer Job (already implemented)

There is a service that synchronizes the producer and consumer databases using a batch approach (see Fig. 10).

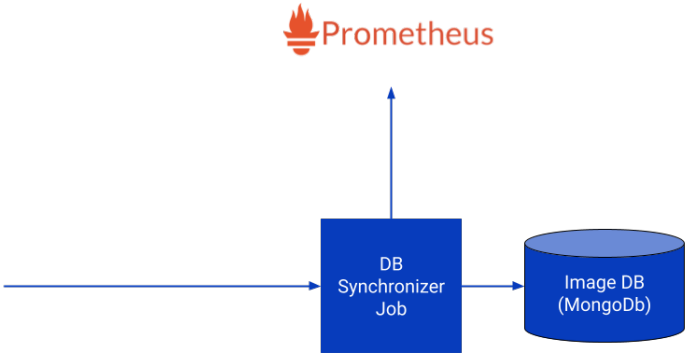

Figure 10: Db Synchronizer Job - UI

This job has a UI (see Fig. 11) located at:

```
{
http://localhost:5003/ - locally
db-synchronizer-job.leaf-image-management-system.svc.cluster
.local:5000 - kubernetes
}
```

Figure 11: 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.

This service also has a Prometheus port (see Fig. 12):

```
{
http://localhost:8051 - locally
db-synchronizer-job.leaf-image-management-system.svc.cluster
.local:8050 - kubernetes
}
```

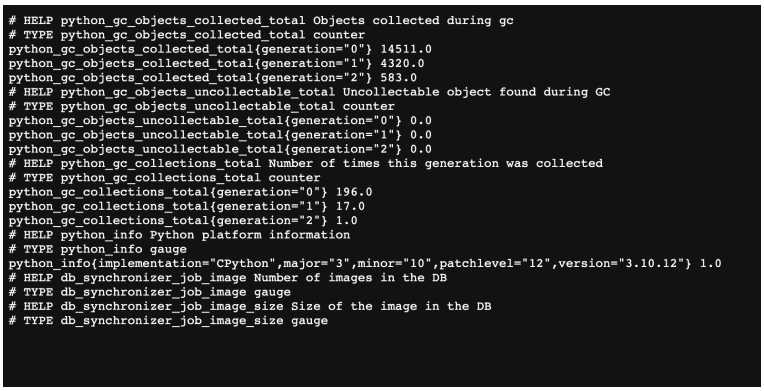

Figure 12: Db Synchronizer Job - Prometheus

Prometheus allows you to retrieve metrics of the service, which are used by Grafana. For the Db Synchronizer Job, it counts the number of received images as well as the size of messages transmitted to the service.

6.4 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. 13).

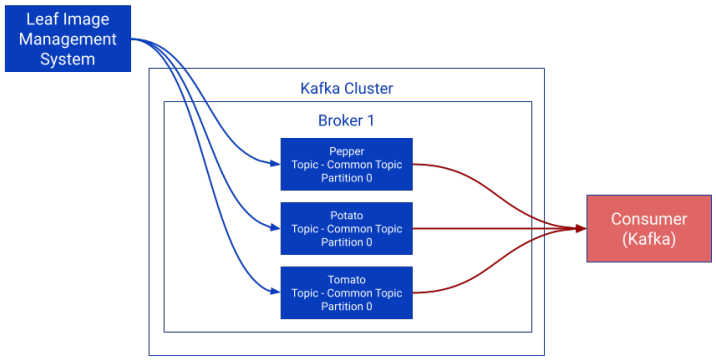

Figure 13: Kafka cluster - Schema

It's recommended to use Conductor (refer to Fig. 14) to display the topics and monitor production.

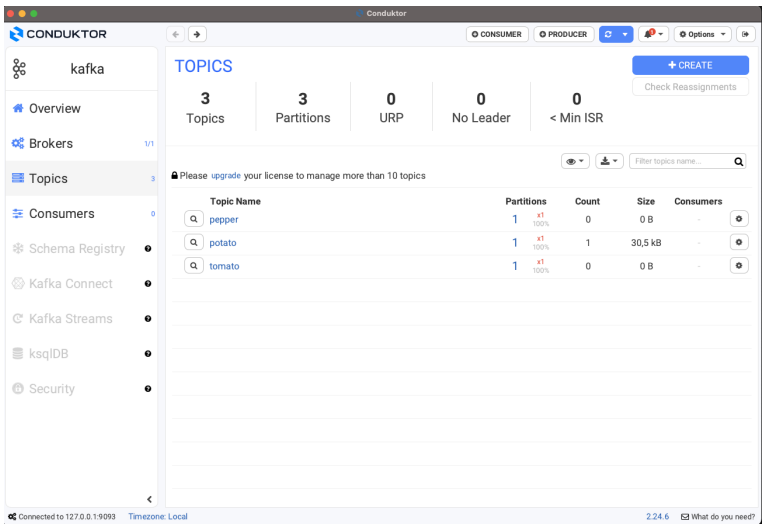

Figure 14: Kafka cluster - Conductor

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
}
```

6.5 Grafana (aleady implemented)

THIS SECTION OF THE ASSIGNMENT REMAINS INCOMPLETE.

6.6 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 consumer Db to the producer Db. 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. 15). The main task is to consume at least one topic from the Kafka cluster and create, update, or delete information in the consumer Db for the producer Db.

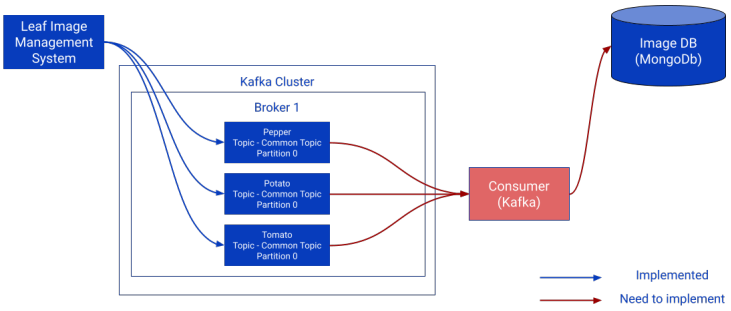

Figure 15: Consumer - Schema

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 this with a single application.

When you have deployed the service alongside the entire system, you must conduct an 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.

7 Infrastructure

Every service in this schema operates within a network named 'leaf-image-management-system'. You need to use this schema for the consumer. Also, you must obtain the already-implemented services. The instructions and additional resources can be found at `add_link_to_gitlab`.

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 above repository:

```
start.dev.sh
stop.dev.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.dev.sh script. All services have a single instance.

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
stop.prod.sh
```

Remember that you can always stop all jobs in the service and prevent the creation of new images, as well as possible updates to already created ones.

Some services might be slower than others, especially the deployment of databases. This is 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.

8 Resources

These resources will help you to understand the question deeply, as well as to 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/
- 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/
- PyMongo (library to use MongoDB with Python) – https://pymongo.readthedocs.io/en/stable/