Cloud Computing Assignment 2 (Additional Description)

Scientific Computing Research Group, University of Vienna

1 Overview

This description covers the services already implemented in the assignment, helping you better understand the created Leaf Image Management System, as well as the data and metrics for the service.

2 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. 1 illustrates that each type has its own set of diseases, where each disease constitutes a class. For example, potato leaves encompass three classes.

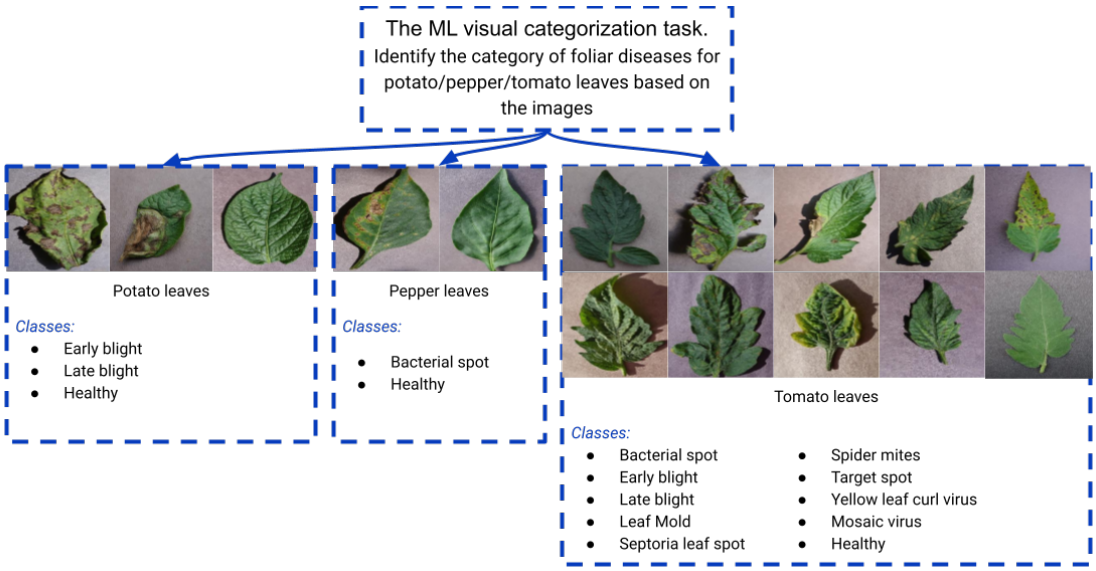

Figure 1: Leaf Diseases Classes

3 Detailed APIs' Description

3.1 Image API (already implemented)

The description about this service is represented in the main assignment file. However, you can 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 (content-type: application/json):

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

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

2. **Check if the producer database is deployed**

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

   ```
   [GET] /image-plant/is-deployed
   ```

   Response provides the following information if the producer database is deployed (content-type: application/json):

```
{
  "is_deployed": true
}
```

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

3. **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.

4. **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.

5. **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 third 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.

6. **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.

7. **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 third 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.

8. **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 third 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.

9. **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. 2) that can be accessed via:

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

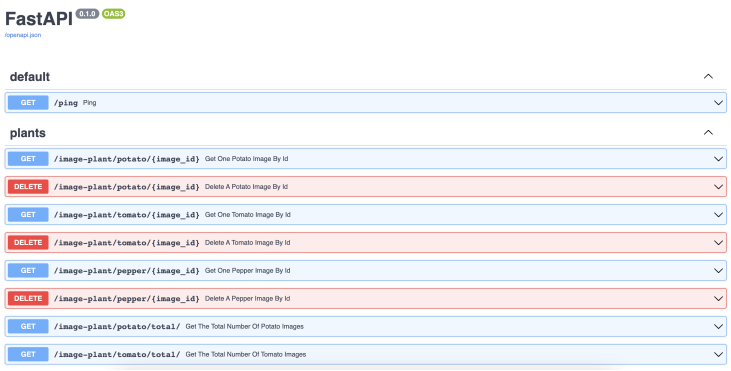

Figure 2: Image API - Swagger

3.2 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 checking if the service works:

   ```
   [GET] /ping
   ```

   Response provides the following information if the service is operational (content-type: application/json):

   ```
   "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 a specific type of leaf (content-type: application/json):

```
{
  "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 (content-type: application/json):

```
{
  "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. 3), which can be opened by:

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

Figure 3: Image Analyzer API - Swagger

4 Mongo Database for producer/consumer description

We use MongoDB to store data, and it is deployed and populated automatically alongside other services. As we described, the entire schema contains two databases: Producer Image DB (mongodb-producer), Consumer Image DB (mongodb-consumer).

For the producer, you can open the database using the following address:

```
{
    http://localhost:27017 - locally
    mongodb-producer.leaf-image-management-system.svc
    .cluster.local:27017 - on Kubernetes
}
```

For the consumer, you can open the database using the following address:

```
{
    http://localhost:27018 - locally
    mongodb-consumer.leaf-image-management-system.svc
    .cluster.local:27017 - on Kubernetes
}
```

5 Prometheus and Grafana (already implemented)

5.1 Prometheus for Image API

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.

The Image API provides an open Prometheus port (see Fig. 4):

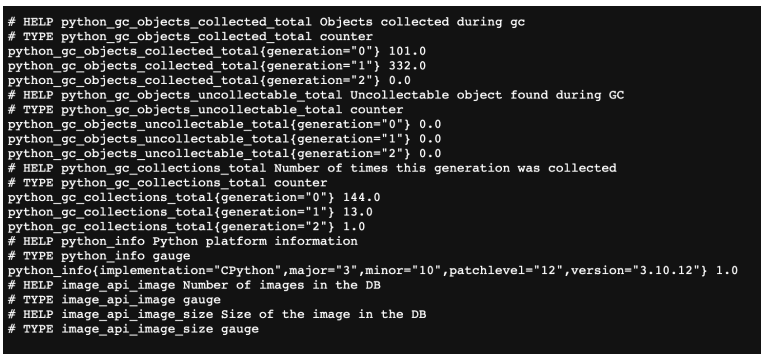

Figure 4: Image API - Prometheus

Prometheus uses the following port:

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

For Image API we gather the following metrics:

```
{
    image_api_image - counter of images
    image_api_image_size - size of the image in bytes
}
```

5.2 Prometheus for DB Synchronizer Job

This service also has a Prometheus port (see Fig. 5):

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

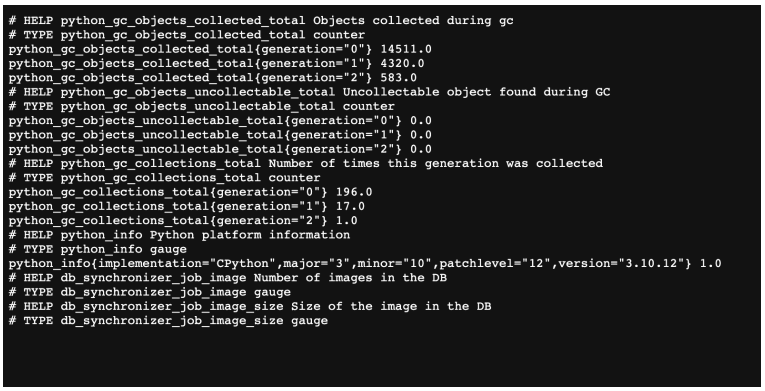

Figure 5: DB Synchronizer Job - Prometheus

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.

For DB Synchronizer Job we gather the following metrics:

```
{
    db_synchronization_job_image - counter of images
    db_synchronization_job_image_size - size of the image in
    bytes
}
```

5.3 Prometheus and Grafana deployment

After implementing your consumer, you will need to use Prometheus to address Task 3, which involves comparing the bandwidth between the pre-implemented batch processing application and the stream processing application. The Prometheus cluster is deployed separately, and details about it can be found in the README file. We use a new namespace named 'metrics' to deploy the Prometheus cluster.

The Prometheus cluster gathers metrics from the Prometheus services associated with ImageAPI and DB Synchronizer Job. To access the Prometheus cluster, you need to connect to the following addresses:

```
{
    http://localhost:9090 - locally
    prometheus-kube-prometheus-prometheus.metrics.svc
    .cluster.local:9090 - kubernetes
}
```

Prometheus features a UI located in the aforementioned address as shown in Fig. 6.

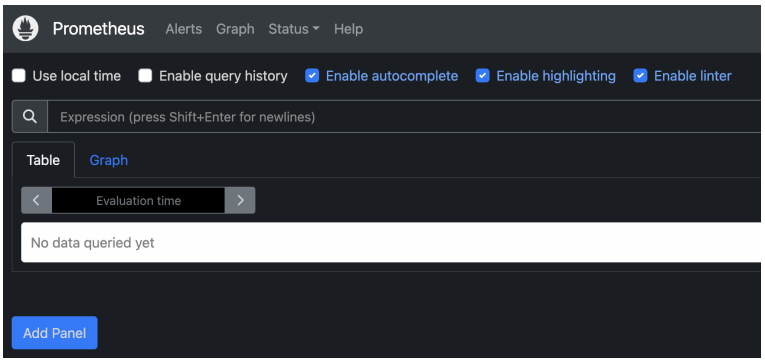

Figure 6: Prometheus - UI

While 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 Prometheus in the 'metrics' namespace and also has its own UI (Fig. 7).

Figure 7: Grafana - UI

6 Google Cloud Platform (GCP) and Google Kubernetes Engine (GKE)

This brief tutorial will help you create a new project and prepare for deploying LIMS to GCP. First and foremost, you need to create a project. You can find the 'Create a Project' page by using the search area (Fig. 8).

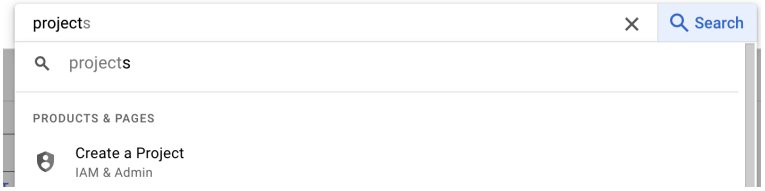

Figure 8: Grafana - UI

Next, you need to enter the name of the project and click 'Create' (Fig. 9).

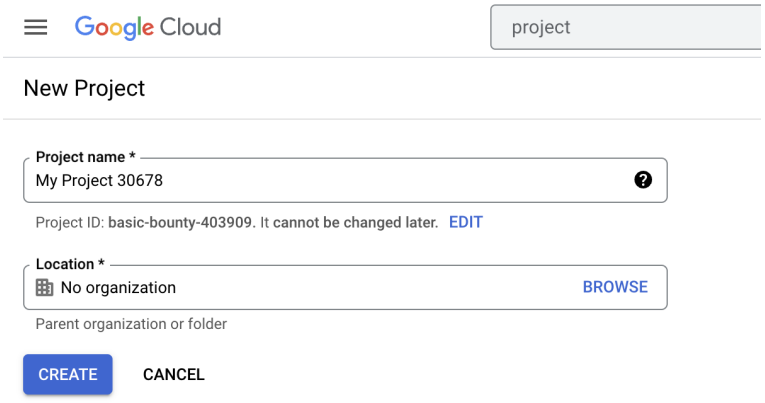

Figure 9: Grafana - UI

The result can be seen in Fig. 10.

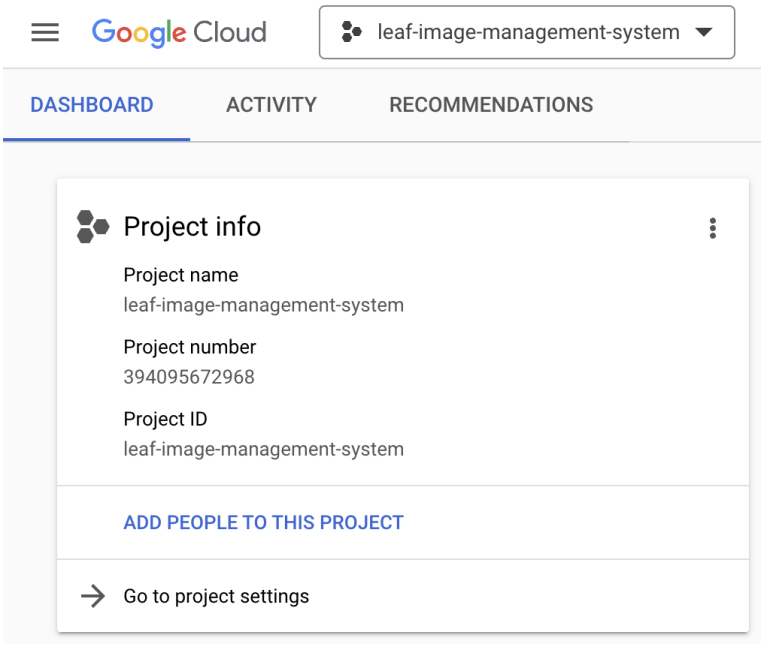

Figure 10: Grafana - UI

Following that, you need to install gcloud locally and then authenticate using:

```
gcloud auth login
```

The result can be seen in Fig. 11.

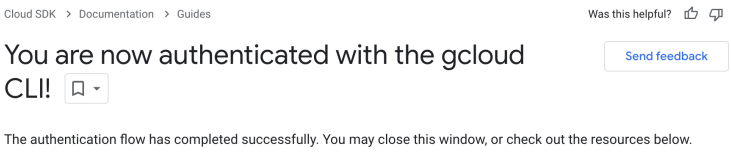

Figure 11: Grafana - UI

Then, locally, you need to set your created project:

```
gcloud config set project PROJECT_ID
```

You need to enable the Kubernetes Engine API for your project:

```
gcloud services enable container.googleapis.com
```

You need to create a cluster specifying the machine type, number of nodes, and zone:

```
gcloud container clusters \
create leaf-image-management-system-cluster-cpu \
--machine-type n1-standard-4 \
--num-nodes 1 \
--zone us-west1-c
```

As you can see, an n1-standard-4 machine is required to effectively deploy the cluster.

Don't forget to delete the cluster after testing. For this purpose, you need to use this command:

```
gcloud container clusters \
delete leaf-image-management-system-cluster \
--zone us-central1-c
```

To deploy the LIMS and metrics, you need to execute the following shell script files located in the repository:

```
start.prod.sh your_project_id (from GCP)
start-metrics.prod.sh
```