Assignment 1 Overview

In this exercise, you will need to develop services that reside within a bigger application. All services need to communicate with each other regardless of their implementation language or framework. In the description below, all of the services are listed, but you need to implement only those that are explicitly requested.

This assignment consists of two tasks:

1. [87%] Developing a functional system comprised of many services, where you need to:
    - Develop 2 services that work with the rest of the provided services
        - You can develop the system using Docker and Compose
        - Services must be able to communicate between each other
        - Have resource constraints defined using Docker Compose / Kubernetes
        - System should tolerate failures to some extent
          i.e., if 1 request fails, it should not crash to whole system
    - Test and run using Google Kubernetes Engine (GKE)
        - On GKE the only entry point should be through an Ingress, which should forward requests to your services
        - It should be possible to scale services using Kubernetes
        - Deliver Kubernetes manifest files
        - Spend up to $10 (always shut down the cluster after testing!)
    - Use Cloud Run for Serverless Containers (or functions) for the HumanDetection service

2. [13%] Estimating costs using GCP Pricing Calculator:
    - Choose an Infrastructure for 1 year for three different scenarios (pick a cluster size based on your estimate needed for your system):
        a. **Standard GKE cluster**
           Discuss the number of E2 machines you need, and what type with respect to the resource usage described in your deployment files (e.g., the number of E2 machines with 2vCPUs cores and 8GB RAM).

        b. **Autopilot GKE cluster**
           Compare to (a) GKE prices for a year, use the same number of vCPU as for the standard cluster (you don't need to deal with both clusters just calculate the pricings). Are there price differences and what are the pros and cons?

        c. **Google Cloud Run**
           Compare the pricing to what you would need to pay for Google Cloud Run. You can ignore the eventual storage requirements you may need, and try calculate how many requests per month you would need to have to reach to have roughly the same costs as with the previous two. Select 1 CPU, 512MB of memory per container, 500ms execution time per request, minimum 1 number of instances to be kept "warm".

Deliverables

Your work results should include the following:

1. Source code of your services (on our Gitlab)

   - Submissions should be pushed to your repository on our Gitlab Server in A1 folder.
   - Place sources of each service in a separate folder, a **Dockerfile** should be included
   - Write one **Docker Compose** for all services
     - with resource constraints for each service
     - two networks (backend and cameras)
   - Kubernetes manifest files written in YAML and present in the **/manifests** folder

2. Documentation (A PDF in Moodle (or README.md on our Gitlab)):

   The document should consider the following:
   - Implementation details (networking, setup, and other relevant aspects)
   - Discuss problems you have encountered during the development
   - How could your application be improved
   - Scalability, Bottlenecks (after testing on a Kubernetes cluster)
     - Is your application scalable? How? The whole system or only some services?
     - Which ones are the most beneficial services to scale?
     - What happens when the number of cameras increases; how will your code handle multiple streams?
       - e.g., use >4 cameras and scale services, can you process more images at the same time by scaling some of the services?
       - Is there anything that hinders scaling, and can it be improved?
     - Could some parts of the system be improved to support better scaling?
     - Describe how you scaled your application on GKE, and with how many replicas
   - Service discovery: How are services able to see each other in your setup?
   - Communication / Data Flow
     - What information a service receives from other services, and what does it send to other services? Are there other ways?
   - What are the most computationally intensive services? (Remember to specify resource constraints in your configuration files (compose/Kubernetes manifests)
   - Types of Kubernetes objects you used (deployments, services, pods, …), for what purposes, and what kinds of services (NodePort, ClusterIP, or LoadBalancer)
   - Ingress, setup, type, and to which services are requests forwarded
   - What did you use for the development, and what did you need to change to enable execution on the GKE? (if you did not develop directly on GCP)
   - Configuration of your cluster on GKE (if different from specification)
   - Discuss yearly costs of provisioning the full cluster. How do you think it compares to buying and maintaining your own hardware (use just rough estimates)?
   - Resource usage, how many vCPUs and memory your system required, and how does this compare to your calculation?
   - Include screenshots of your GKE Cluster, Workloads, Services & Ingress on GCP.

   Document things that you found interesting, challenging, or problematic and push to your private git repository (typically https://gitta-lab.cs.univie.ac.at/cloud-computing-2022w/a<MNr>).

Task Description

The airport has undertaken a restructuring program and invested significantly in expanding its capacity, building new runways and terminals, following up with a comprehensive marketing campaign. This resulted in an increased number of flights as expected. However, apart from the core business investments, the airport failed to follow suit with their aging IT infrastructure due to limited funding.

Their airport software was engineered years ago as a complex, monolithic application and is maintained by a small team of airport employees. When the increasing number of flights and passengers caused increased workloads, the system struggled to keep up. Limited scaling options were shown viable for a monolithic application, so server hardware upgrades made little impact. Furthermore, it was hard to keep up with new requests for changes, such as implementing evolving regulatory frameworks, patching policies, or keeping up to date with recent technological developments. Consequently, system downtime and staff overtime are common, so it became clear that the system needs some rethinking to serve the business requirements at the level the company has scaled to.

The airport conducted a small-scale analysis to identify upgrades that address the most critical services and minimize the time needed to achieve meaningful improvements. Keeping in mind under-capacitated IT staff, the development has been divided into several contracts to expedite development. Each contract covers a group of related functionalities, and all will be executed simultaneously by different companies. One of those contracts has been awarded to you.

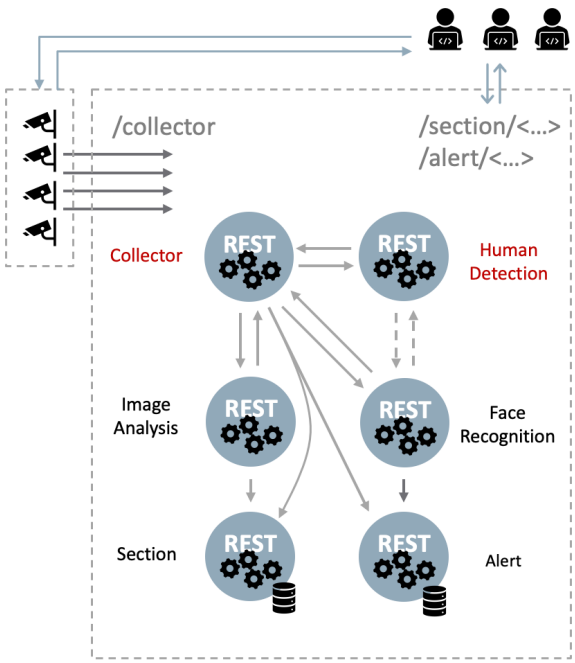
The focus of the first phase of your contract is airport security. Airport's surveillance cameras are placed on each entrance or exit to track the number of people passing through different sections of the airport terminal. You have received the following set of preferences:

- They prefer having several smaller, more manageable services rather than a bigger monolithic application.
- The existing REST interfaces should be preserved
- They decided to containerize their services, and they would like to use Kubernetes

Detailed Description

Images from surveillance cameras are used to collect statistical data on how many people passed through a particular gate, so they can put additional staff to better support people on gates and improve overall security in crowded areas. Each captured image (including metadata, like originating section, timestamp, etc.) needs to be sent to internal services for further analysis. The goal is to have statistical data on how many people (identified only by estimated age and gender by the ImageAnalysis service) passed through a particular section of the airport in a given timeframe. Additionally, they want to be able to identify persons of interest and to be able to determine if they entered the airport area by using FaceRecognition service against a list of known persons of interest.

The current prototype includes the following services (details in subsequent sections):



- Camera: represents a surveillance camera, capturing images and sending them to the system for analysis.
- ImageAnalysis: takes an image, attempts to identify human faces, and tries to estimate their age and gender.
- FaceRecognition: compares a given image against the list of persons in a predefined database for possible matches.
- Section: manage and provide statistical data for a section.
- Alert: manages and provides information about detected persons of interest.
- HumanDetection: takes an image, attempts to detect the presence of human beings, and forwards the request if needed.
- Collector: mainly handles communication flow, e.g., receives images from Camera services and forwards them for processing to other services.

Camera, FaceRecognition, ImageAnalysis, and Alert services have already been implemented. You need to implement Collector, and HumanDetection services. Functional requirements are given below.

Camera Service (already implemented)

The camera services simulate the surveillance cameras at the airport. Each produced image signifies an event triggered when motion has been detected (e.g., a person entering or exiting a particular airport section). When switched on, this service stream images (frames) to a given destination (URL). Captured images have two different representations:

Frame Object - this object stores the base64-encoded imaged data, the timestamp when the image was captured, ID of the section in which the photo was taken, and whether the person was entering or exiting the area when the photo was taken

Raw Image - A plain raw JPEG image data with no other info attached ("image/jpeg"), used for testing.

Remark: Please note that since we do not have real cameras in this exercise, this service uses a dataset of human faces collected over the internet (see more http://vis-www.cs.umass.edu/fddb/). The Camera service picks up images from this dataset and sends them to the destination service, using POST requests.

The Camera service API specification is as follows:

[Base URLs: http://localhost:311xx (host network), http://camera (container network]

POST /stream?toggle=on

Turns ON or OFF streaming to the specified destination (URL). This functionality will send Frame objects in the JSON format to the destination. If the toggle is not supplied, "on" is the default value. The "max-frames" value limits the number of frames that are sent to the destination. If it is not supplied, all available images are sent. The delay field is used to specify a delay in seconds between subsequent requests (default is 0 or no delay).

Body ("application/json"):

```
{
    "destination": "<destination-url>",
    "max-frames": 10, (optional)
    "delay": 0.1, (optional)
    "extra-info": ""  (optional)
}
```

Response ("application/json"):

```
{
    "code": 207,
    "type": "STATUS",
    "message": "Streamed 1 frame(s) in x seconds, with 0 failed requests."
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

GET /config

Returns the current configuration of the Camera service.

Response ("application/json"):

```
{
    "section": 1,
    "event": "entry",
    "name": "Camera X" (optional)
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

PUT /config

Configures the Camera service according to the configuration provided

Body ("application/json"):

```
{
    "section": 1,
    "event": "entry",
    "name": "Camera X" (optional)
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

GET /frame (primary for testing purposes)

Captures an image and returns it in a JSON object or as a raw image to the requestor. You can use HTTP headers to switch between the types.

Response (if the request header has Accept: "application/json"):

```
{
    "timestamp": "2020-10-24T19:28:46.128495",
    "section": 1,
    "event": "exit",
    "image": "<base64-encoded-string>",
    "extra-info": ""
}
```

Response (if the request header has Accept: "image/jpeg"):

```
raw image, with "image/jpeg" content-type
```

Codes: 2xx if the operation was successful, otherwise 4xx.

ImageAnalysis (already implemented)

This service analyzes a given image for human faces and tries to estimate the age and gender of people detected on the image (this service cannot recognize a person). The collected JSON data is sent back as a response or forwarded (POST) to the specified destination.

The ImageAnalysis service API specification is as follows:

[Base URLs: http://localhost:312xx (host network for testing), http://image-analysis (container network)]

POST /frame

Accepts a JSON with base64-encoded image for analysis and returns a list with estimated information on detected persons.

Body ("application/json"):

```
{
    "timestamp": "",
    "image": "<base64-encoded-string>",
    "section": 1,
    "event": "entry",
    "destination": "<destination-url>", (optional)
    "extra-info": "" (optional)
}
```

Response ("application/json"):

```
{
    "timestamp": "2010-10-14T11:19:18",
    "section": 1,
    "event": "entry",
    "persons": [
        {
            "age": "60-100",
            "gender": "Male"
        }
    ],
    "extra-info": "", (optional)
    "image": ""<base64-encoded-string>"
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

POST /frame (raw image)

Receives an image for analysis and returns a list with estimated information on detected persons.

Request Headers: Content-Type: "image/jpeg",
Body ("application/json"):

```
raw image, with "image/jpeg" content-type
```

Response ("application/json"):

```
{
    "persons":[
       {
        "age": "8-12",
        "gender": "female"
       }
    ]
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

Remarks: If the "destination" field is provided, the result is forwarded to the destination.

FaceRecognition (already implemented)

This service compares the received image against the predefined database of persons of interest for possible candidates.

The FaceRecognition service API specification is as follows:

[Base URL: http://localhost:313xx (host network for testing), http://face-recognition (container network)]

POST /frame

Receives an image for analysis in JSON format and returns a list with information on detected persons in JSON. If the "destination" field is provided, the result is forwarded (POST) to the destination.

Request Body ("application/json"):

```
{
    "timestamp": "2010-10-14T11:19:18",
    "image": "<base64-encoded-string>",
    "section": 1,
    "event": "entry",
    "destination": "<destination-url>", (optional)
    "extra-info": "" (optional)
}
```

Response ("application/json"):

```
{
    "timestamp": "2010-10-14T11:19:18",
    "section": 1,
    "event": "entry",
    "image": "<base64-encoded-string>",
    "known-persons": [
        { "name": "Mr. Y" }
    ],
    "extra-info": "" (optional)
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

POST /frame (raw image)

Checks if the provided image contains any known persons of interest. Accepts raw JPEG image data and returns a list of detected persons' names.

Request Headers: Content-Type: "image/jpeg"
Request Body:

```
raw image, with "image/jpeg" content-type
```

Response ("application/json"):

```
{
    "known-persons": [
        { "name": "Ms. X" }
    ]
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

Remarks: It may take a while until you detect some person of interest. In case the "destination" field is provided, the result is forwarded to the destination.

Section (already implemented)

This service takes care of sections' statistical data. The list of all persons that passed through a section of the airport is kept in a data storage. By default, each Section service stores its data in a local mongo database. If you decide to use a single database for all Section service instances, then you can configure that with the following environment variables:

EXTERNAL_MONGODB_URL=<mongodb://<hostname:port/>
EXTERNAL_MONGODB_DB=<db-name>
EXTERNAL_MONGODB_COLLECTION=<collection-name>

The Sections service API specification is as follows:

[Base URLs: http://localhost:314xx (host network), http://section (container network)]

POST /persons

Creates a new entry in the list of detected persons passing through this section.

Body ("application/json"):

```
{
  "timestamp": "2010-10-14T11:19:18",
  "section": 1,
  "event": "exit",
  "persons":[
      {
        "age": "8-12",
        "gender": "male"
      }
  ],
  "image": "<base64-encoded-string>", (optional)
  "extra-info": "" (optional)
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

GET /persons?from={from}&to={to}&aggregate=count&event=exit&section=1

Returns a JSON with a list of persons in this section in a given time frame (use below specified time format). Additionally, it allows aggregate=count for returning the total number of persons in the section, aggregate=time for returning the total time between the first and the last received event, and filtering by event type. Filtering by section number is optional and used only when all Section service instances are using the same database.

Response ("if query is: ?from={from}&to={to}&event=entry"):

```
{
  "persons":[
      {
        "uuid": "0aa57c62-68db-4d38-bdd2-5ea47dds3d2e",
        "age": "8-12",
        "gender": "male",
        "timestamp": "2010-10-14T11:19:18.039111",
        "section": 1,
        "event": "entry",
        "extra-info ": "" (optional)
      }
  ]
}
```

Response ("if query is: ?from={from}&to={to}&aggregate=count"):

```
{
    "count": 42
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

Alert (already implemented)

This simple service handles alerts in the system. Alerts represent detected persons of interest.

The Alert service API specification is as follows:

[Base URLs: http://localhost:315xx (host network), http://alert (container network)]

POST /alerts

Creates a new alert by providing an estimated age and gender of a person, as well as the image of the face along with the timestamp when it was captured.

Body ("application/json"):

```
{
  "timestamp": "2010-10-14T11:19:18",
  "section": 1,
  "event": "exit",
  "known-persons":[
      {
        "name":"Ms. X"
      }
  ],
  "image": "<base64-encoded-string>", (optional)
  "extra-info": "" (optional)
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

GET /alerts?from={from}&to={to}&aggregate=count&event=entry

Returns a JSON with a list of alerts in a given time frame. Additionally, it allows aggregate=count for returning the total number of created alerts and filtering by event names.

Response ("if query is: /?from={from}&to={to}"):

```
{
  "alerts":[
      {
        "uuid": "5aa57c62-66db-4d38-bdd2-5ea37dd9352e",
        "timestamp": "2010-10-14T11:19:18.039111",
        "known-persons":[
                {
                   "name": "Mr. Y"
                }
        ],
        "section": "1",
        "event": "entry",
        "image": "<base64-encoded-string>", (optional)
        "extra-info ": "" (optional)
      }
  ]
}
```

Response ("if query is: /?from={from}&to={to}&aggregate=count"):

```
{
    "count": 42
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

GET /alerts/{uuid}

Return the alert details, i.e., person's info, timestamp, event, section, and the corresponding image.

Response ("application/json"):

```
{
  "uuid": "5aa57c62-66db-4d38-bdd2-5ea37dd9352e",
  "timestamp":"2010-10-14T11:19:18",
  "section": 1,
  "event": "exit",
  "known-persons":[
      {
         "name":"Ms. X"
      }
  ],
  "image": "<base64-encoded-string>", (optional)
  "extra-info ": "" (optional)
}
```

Codes: 2xx if the operation was successful, otherwise 4xx.

DELETE /alerts/{uuid}

Removes the specified alert.

Codes: 2xx if the operation was successful, otherwise 4xx.

HumanDetection (to be implemented)

This service analyzes a given image for presence of humans using Google Cloud Vision API. The service takes a base64-encoded image in a JSON object (received via POST request) and sends it for object detection using Google Cloud Vision API. The service should then analyze the response from the Google Cloud Vision API; If a person is detected, the response JSON from GCP will contain an

entry describing the detected entity as a person. For example, when the service receives an “HTTP POST /frame” request with a base64 image included in the JSON payload, it then makes a request to https://vision.googleapis.com/v1/images:annotate with the following content:

```
{    "requests": [
        {
            "image": {
                "content": <forwared-base64-image>
                    "image"
                ]
            },
            "features": [
                {
                    "maxResults": 10,
                    "type": "OBJECT_LOCALIZATION" (or "FACE_DETECTION")
                } ] } ] }
```

The service then analyzes the response from Google API for a “Person” entry, which looks like this:

```
{    "responses": [ {
            "localizedObjectAnnotations": [
               {
                    "boundingPoly": { ... },
                    "mid": "/m/01g317",
                    "name": "Person",
                    "score": 0.8794785
                },
                ] }
        ]
}
```

If your service detects such entry, it should add that info to the originally received JSON by adding a simple additional field (**'person-count'**) in the response JSON data. This is the minimum functionality of this service. However, it can also be used to interact with other services in the system as needed, and its API can be extended to fit your solution's.

The Google Cloud Vision endpoint where the service sends a request with a base64 image to is:

> The **https://vision.googleapis.com/v1/images:annotate**?key=<your-api-key>

For this assignment, you can hard-code the API key value in your implementation or pass it via an environment variable. You can find out how you can use the Google Cloud Vision API here, along with an example of a JSON object that you need to send:

> **https://cloud.google.com/vision/docs/object-localizer#vision_localize_objects-drest**

The HumanDetection service API specification is as follows:

[Base URLs: http://human-detection (container network)]

POST /frame

> Receives an image for analysis in JSON format, and if persons are detected, it adds the detected person count, thus signifying that further analysis is required.
>
> Body ("application/json"):
>
> ```
> {
>     "timestamp": "2010-10-14T11:19:18",
>     "image": "<base64-encoded-string>",
>     "section": 1,
>     "event": "entry",
>     "destination": "<destination-url>", (optional)
>     "extra-info": "" (optional)
> }
> ```

Response ("application/json"):

```
{
    "timestamp":"2010-10-14T11:19:18.039111Z",
    "section": 1,
    "event": "entry",
    "image": "<base64-encoded-string>",
    "person-count": 1, (added field!!)
    "destination": "<destination-url>", (optional)
    "extra-info": "" (optional)
}
```

Response ("if destination is provided"):

```
remote service response or just a code
```

Codes: 2xx if the operation was successful, otherwise 4xx.

Collector (to be implemented)

This service mainly handles the communication flow in the system (at least in part). For example, when it receives a frame (through POST **/frame**), this service could (depending on your design decisions) forward this image for analysis to the ImageAnalysis service, which returns estimated person's age and gender, then transform this data if necessary and send it in the corresponding format to the Section service (see the Section service description for /persons) for storage of statistical information. Additionally, the service could also send the same data to the FaceRecognition service, which will try to identify persons of interest. Note that if the "destination: <URL>" field is sent to the FaceRecognition or ImageAnalysis service, these services will try to send the result to the specified destination; otherwise, the result will be returned in response to the original requestor, which is the Collector service in this case. This service may interact with all or some of the other services, depending on your design decisions.

The Collector service API specification is as follows:

[Base URLs: http://collector (container network)]

POST /frame

Adds a new frame to the Collector service.

Body ("application/json"):

```
{
    "timestamp": "2010-10-14T11:19:18.03",
    "image": "<base64-encoded-string>",
    "section": 1,
    "event": "entry",
    "destination": "<destination-url>", (optional)
    "extra-info": "" (optional)
}
```

Response ("if destination is provided"):

```
remote service response or just a code
```

Codes: 2xx if the operation was successful, otherwise 4xx.

POST /persons (optional)

Allows the user or another service to push detected person information.

Body ("application/json"):

```
{
  "timestamp": "2010-10-14T11:19:18",
  "section": 1,
  "event": "exit",
  "persons":[ { "age": "8-12", "gender": "male" } ],
  "destination": "<destination-url>", (optional)
  "image": "<base64-encoded-string>", (optional)
  "extra-info": "" (optional)
}
```

POST /known-persons (optional)

Allows the user or another service to push detected information on persons of interest.

Body ("application/json"):

```
{
  "timestamp": "2010-10-14T11:19:18",
  "section": 1,
  "event": "exit",
  "known-persons":[  { "name": "Ms. X" }  ],
  "destination": "<destination-url>", (optional)
  "image": "<base64-encoded-string>", (optional)
  "extra-info": "" (optional)
}
```

Response ("if destination is provided"):

```
remote service response or just a code
```

Codes: 2xx if the operation was successful, otherwise 4xx.

Objective Details

Your objective is to implement specified services, using Docker as containerization technology and any technology of your choice inside the containers. Besides basic communication and a functional system, you should decide how to scale the system or the individual services to better process the streaming data (think about the vertical and horizontal scaling). Written documentation on your design decisions and implementation details is the essential part of this task. You need to elaborate on potential bottlenecks of your system and parts that could be improved and explain why it is so. There are different ways to pass data around the system. Make sure that you include this in your documentation. Describe how you use your Section service(s) and how data is saved and accessed. Why did you choose to do it like this, and could that be better?

The implementation work goes beyond just implementing the specified API. There is more than one way to implement the task, and you may want to use additional back-end or Docker functionalities to organize communication, configure the system, etc. Your decisions should be documented.

Additional remarks:

For this prototype, we assume that the system collects information only for a limited amount of time. Unless specified otherwise, Camera services will stop streaming automatically after all images in the database are streamed (roughly 0.25 GB in 13k images)

For all dates in this exercise, use the following format: "2019-10-12T09:49:14" or "2010-10-14T11:19:18.039111" (without time zone information)

If a response HTTP code is not specified, select an appropriate one

For the implementation of Web Services, you may choose any technology you prefer, as long as it can run in a Docker container and ships with a Dockerfile

The already implemented services may occasionally fail, return errors, or be unresponsive; this is the intended behavior and should be discussed in the documentation, i.e., how to handle errors and how much of the whole system is affected.

In addition to the network configuration, each service should have some resource constraints (limit at least the CPU usage for each service).

Tips: Start with a configuration of the system that runs a single instance of each service. Once your system functions properly, try to reconfigure the system for more instances and observe how your system functions. Already implemented services are single-threaded and will not do any asynchronous calls.

Infrastructure, Docker, Kubernetes

While you can develop your services in a single Docker network, your final solution should use two networks – one for cameras (camera-network), and one for other services (backend-network). Cameras need to be able to stream to the Collector service.

You need to get the already implemented services. The instruction and additional resources can be found on:

git clone https://gitta-lab.cs.univie.ac.at/public-examples/cloud-computing-a1-services-2022w.git

In short, you need to use the following images from DockerHub: ccuni/camera-service-2022w, ccuni/image-analysis-service-2022w, ccuni/face-recognition-service-2022w, ccuni/section-service-2022w, docker pull ccuni/alert-service-2022w.

The readme files will tell you how to get these services started. These are simple, containerized python flask applications that implement the above-specified APIs to some extent. Remember that you need to work with the containerized services, not with the python apps running on the host. The README files will simply tell you to get the container image from the DockerHub and run this container.

It is probably easier to develop the system on your laptop. Start by starting up containerized services run and try to test them out. The services need to be on the same Docker network to communicate with each other. For this purpose, use docker-compose. You can start from the provided docker-compose.yml provided in the repository, but you need to add two of your services. For testing you can manually create a Docker network and assign net-aliases (DNS names) for all services, but it is usually easier to use docker-compose directly. Try to develop your services in the container, as you will not be able to communicate with other services otherwise (unless you use the host network for development purposes, but that is not always available).

Note that a request from a camera **triggers a chain of requests throughout the system**, so if the message is not reaching the final destinations, you will need to figure out where exactly the problem is, and this may not be the service you are sending a request to. Note that all already implemented services operate sequentially.

Some useful docker-compose can be found in the provided repository, essentially you need to call "**docker-compose up**" in the folder where your **docker-compose.yml** resides.

Some services may start slower than others. If your services need to wait for other services to spin up, then you can describe that in docker-compose by specifying the **'depends_on'** field. You can also describe environment variables in docker-compose.

You can find information on Docker resource constraints here:

https://docs.docker.com/config/containers/resource_constraints/

and for Compose:

https://docs.docker.com/compose/compose-file/

Note that docker-compose has its own load balancing capabilities for scaling services (without using Docker Swarm). If you need such capabilities, you can utilize them (in --compatibility mode for Compose), or you can build your own simple ones or utilize some other technologies like nginx and utilize them for this exercise.

Once you get your system running, try to scale it by increasing the number of replicas of your services. You can test scaling with docker-compose with the "—scale <service-name>=<number-of-instances>". You should try to scale your services on the GKE cluster. You can start with 1 instance of each service and >=2 cameras. Try to observe if you gain any performance gains. Note that Section and Alert services also allow you to get time difference between the first and the last request they have received on their side, in a given timeframe and filtered by provided event and section: **?from={from}&to={to}&event=entry&section=1&aggregate=time**. Since alerts can be sparse, you may want use the Section service for this. The time provided by the camera stream may or may not be reliable depending on your design.

When using Kubernetes, you can try to forcibly kill one of the containers running on Kubernetes and observe what happens.

Notes: Since some of your services have paths that are similarly named as the actual service, you may end-up with something like http://<your-ip>/alert/alerts. This is fine in the context of this exercise. Therefore, your schema can be "http://<load-balancer-ip>/<service-name>/<service-specific-route>".

All services also accept a GET request to "/probe", and will return a successful response.

Google Cloud Platform

For the **HumanDetection** service you need to use Google Cloud Platform (GCP) Credentials. Once you receive your username and password, you will need to log in to the Google Cloud Platform and setup

a new Project, enable Cloud Vision API, and create an API key. For this simple exercise you can hard-code your API Key inside your application.

- Login to Google Cloud Platform
  https://console.cloud.google.com/
- Create a Project on the Google Cloud Platform
  https://console.cloud.google.com/projectcreate
- Go to API & Services in the Menu
  https://console.cloud.google.com/apis/dashboard

You can test the Vision REST service by following these guidelines for "REST & CMD LINE":

https://cloud.google.com/vision/docs/object-localizer#vision_localize_objects-drest

Just for testing the service directly you can supply your api key as a query string:

POST https://vision.googleapis.com/v1/images:annotate?key=<your api key>

On GCP go to Billing->Budgets & alerts and create a budget for Cloud Vision API to $10 in the current month.

Guides for GKE: https://cloud.google.com/kubernetes-engine/docs/quickstart
Guides for Cloud Run: https://cloud.google.com/run/docs/quickstarts/build-and-deploy#python

Resources, Tutorials

Check out this git repository web service examples in Python and Jax-RS:

https://gitta-lab.cs.univie.ac.at/public-examples/cloud-computing-extra-materials

Additional resources:

1. Flask: https://www.tutorialspoint.com/flask/index.htm
2. Docker - https://docs.docker.com/get-started/
3. Docker - https://github.com/docker/labs/tree/master/beginner/

Java-based Services (typically takes longer to get started than with python):

4. Jersey documentation - https://jersey.java.net/documentation/latest/jaxrs-resources.html
5. IBM RESTful Web Services: The basics - http://www.ibm.com/developerworks/library/ws-restful/
6. REST Web Service Tutorial, with Setup - https://www.tutorialspoint.com/restful/restful_quick_guide.htm
7. REST Web Service Tutorial with Jersey - http://www.vogella.com/tutorials/REST/article.html
8. JAX-RS Endpoint Lifecycle Explained - https://github.com/stoicflame/enunciate/wiki/JAX-RS-Lifecycle
9. JAXB Tutorial - http://www.vogella.com/tutorials/JAXB/article.html