# Cloud Computing

Assignment 1

## Airport Monitoring System

System Implementation and Analysis Report


**Maitham Al-rubaye**

**A12142043**


2023

## Table of Contents

## 1. Introduction

The service is implemented in Python, utilizing the Flask framework for RESTful API interactions and the Pydantic library for data validation. This ensures that the Collector service not only maintains compatibility with existing REST interfaces but also embraces containerization with Docker and orchestration via Kubernetes. The service's logging mechanism is accurately designed to record all operational data, thereby providing transparency and aiding in debugging and monitoring.

The design logic of the Collector service revolves around processing incoming frames from surveillance cameras, enriching them with metadata, and subsequently posting this data to other services for analysis. The initial data parsing is structured to efficiently forward relevant data to the Image-Analysis-Service and Face-Recognition-Service. Following their respective analyses, the results are routed to the Section and Alert services, effectively creating an integrated data flow system that provides real-time analytics on the airport's operations.

The service API, encapsulates functionalities such as error handling, asynchronous requests handling, and response aggregation, ensuring a flexible system that can gracefully handle the incoming demands and potential service disruptions. The Collector service's API is a reflection of a system designed for scalability, robust error handling, and seamless integration within a sophisticated cloud computing landscape.

## 2. Implementation Details

The Collector service is designed as an orchestrator within a distributed microservices architecture. It acts as a middleware that receives image frames from camera services and routing these to the appropriate analytical services (Image-Analysis and Face-Recognition) and management services (Section and Alert).

The service is built using Python, Flask for creating the RESTful API endpoints, and the Pydantic library for data validation, ensuring that incoming requests conform to the expected schema. Httpx is used for asynchronous HTTP requests to downstream services, enabling non-blocking I/O operations, a crucial feature for maintaining throughput under heavy load. The service is structured to first validate incoming frames using Pydantic models. Valid frames are logged and then sending:
- All frames are sent to Image-Analysis, and selective frames, based on the business logic, are sent to Face-Recognition.

- The results from Image-Analysis are used to construct a payload for the Section service, which is then responsible for aggregating statistical data.

- The Face-Recognition service's results are forwarded to the Alert service for security checks against known persons of interest.

In Kubernetes, services are discoverable through internal DNS resolution, which allows the Collector service to route requests to other services by their Kubernetes service names, like http://image-analysis-service:8080. This DNS resolution is a part of Kubernetes' service discovery mechanism that simplifies inter-service communication.

**Encountered Problems and Solutions:**

- Ingress Rewriting Issue: Encountering issues with service accessibility via Postman or curl after deployment to Kubernetes was initially a hurdle. The adoption of Ingress with specific rewrite annotations (nginx.ingress.kubernetes.io/rewrite-target: /$2) resolved this by correctly routing external requests to the appropriate service paths.
- Image Service Timeout: The Image-Analysis-service was experiencing timeouts, which was traced back to insufficient resources. Increasing the service's resource allocation in the Kubernetes configuration solved this issue.
- Inter-Service Communication Challenge: The inability to send requests from the Collector to other services was overcome by implementing models to reformat the JSON payloads. This approach streamlined the process of constructing requests that meet the specific API contracts of each service, thus facilitating smooth communication between services.

**Service Interaction:**

Upon successful implementation of the Collector service, images from camera services began to reach the Alert and Section services consistently. This was not possible initially due to differing API specifications. The Collector-Service now acts as a translator, converting incoming data to the expected formats for each destination service. This is a critical design decision that supports the loosely coupled nature of microservices architectures.

## 3. Communication and Data Flow

The Collector service is at the core of the communication and data flow in the airport surveillance system. It functions by receiving frames from Camera-Service and then routes these frames to the Image-Analysis and Face-Recognition services.
**Inbound Communication:** The Collector service receives JSON payloads with the following structure from Camera-service:

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

**Outbound Communication:** Upon receiving a frame, the Collector service sends a subset of the data to both the Image-Analysis and Face-Recognition services:

```
{
"timestamp": "<timestamp>",
"image": "<base64-encoded-string>",
"section": "<section>",
"event": "<event>"
}
```

**Further Data Routing:** The output from the Image-Analysis-Service, which includes estimated age and gender, is sent to the Section-Service (according to section number) for statistical aggregation. The output from the Face-Recognition-Service is sent to the Alert-Service to notify about persons of interest.

**Communication Method:** The Collector service communicates synchronously with the other services. This means it waits for a response from the Image-Analysis and Face-Recognition services before proceeding. While this approach ensures that the process flow is easily traceable, it may lead to bottlenecks if any service in the chain becomes a latency point.

**Alternatives and Design Decisions:** An alternative design could employ an asynchronous message broker (like RabbitMQ or Kafka) to handle the communication. However, this was not chosen due to the current design of other services, which expect direct HTTP calls rather than subscribing to topics in a message broker. Implementing a message broker would have required substantial changes in the architecture and coordination with other teams responsible for different services.

**Performance Considerations:** The synchronous nature of communication could impact the system's performance, especially under high load. The service could be improved by “Asynchronous Communication”. By switching to an asynchronous model where the Collector-Service publishes messages to a queue and different services consume these messages, we can decouple the services more effectively. This would allow services to operate independently without waiting for responses, potentially improving the system's overall performance.

**Load Balancing with ClusterIP:** The integration of ClusterIP services to facilitate internal load balancing. This approach ensures that traffic to any given service is evenly distributed across its multiple pod instances. By utilizing ClusterIP, it can prevent any single pod from becoming a bottleneck.

## 4. Computational Intensity and Resource Constraints

The most computationally intensive services within this architecture, according to the Grafana monitoring data, are (more CPU usage) Face-Recognition and Image-Analysis services respectively, while (more Memory Usage) Image-Analysis and Face-Recognition services respectively. To manage the resources efficiently in Kubernetes, I defined resource requests and limits for each container. Requests are what the container is guaranteed to get and limits are the maximum resources a container can use. This ensures that a container will never be starved for resources and also prevents a single container from consuming excessive resources, which could impact other services running on the same cluster. Here is an example of how I specify these constraints in Kubernetes deployment YAML for a computationally intensive service like the Face-Recognition-Service:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: face-recognition-service-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: face-recognition-service
  template:
    metadata:
      labels:
        app: face-recognition-service
    spec:
      containers:
      - name: face-recognition-service
        image: gcr.io/univiecca1/face-recognition-service-2023w
        ports:
        - containerPort: 8082
        resources:
            requests:
              cpu: "500m"
              memory: "500Mi"
            limits:
              cpu: "800m"
              memory: "800Mi"
```

These values are chosen based on the observed average usage. I have arrived at these figures through iterative testing and monitoring, increasing the resources if the services are throttled and decreasing if the resources are underutilized, striking a balance between performance and resource efficiency. For the other services I have implemented the same structure but some of them with different resources (as illustrated in snapshots). Moreover, I implemented HPA as explained in section 9.

## 5. Kubernetes Objects and Services

Within the Kubernetes architecture for the system, several core objects are utilized to create a functional and reliable deployment.

- **Deployments:** are used to define the desired state of the application. They manage the deployment and scaling of a set of Pods and ensure that the number of actual replicas meets the desired state specified in the deployment configuration.
- **Services:** are an abstraction layer that defines a logical set of Pods and a policy by which to access them. This abstraction enables Pod-to-Pod communication and provides a single point of entry for accessing the Pods. Services in the service configuration are set up as follows:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: camera-service
spec:
  type: ClusterIP
  selector:
    app: camera-service
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

In this, the camera-service is exposed internally within the cluster using ClusterIP.

**Pods:** They are used to run the application instances. In my setup, Pods are encapsulated within Deployments for ease of management.

```yaml
template:
    metadata:
      labels:
        app: face-recognition-service
    spec:
      containers:
      - name: face-recognition-service
        image: gcr.io/univiecca1/face-recognition-service-2023w
        ports:
        - containerPort: 8082
        resources:
            requests:
              cpu: "100m"
              memory: "100Mi"
            limits:
              cpu: "500m"
              memory: "500Mi"
```

**Ingress:** Ingresses provide HTTP and HTTPS routing to Services from outside the cluster. They are used to provide a unified entry point for the cluster's external traffic.

**Horizontal Pod Autoscaler:** The HPA automatically scales the number of Pods in a deployment, replication controller, replica set, or stateful set based on observed CPU utilization or other select metrics.

## 6. Scalability and Performance

The application is built with scalability in mind, using Kubernetes' inherent ability to handle increased loads by scaling services horizontally. The scalability of individual services allows for a more efficient resource utilization and enhanced performance during demand increasing. While the entire system is scalable, the most beneficial services to scale are typically those that are resource-intensive and those that have to handle the majority of the workload. In this case, the **Face-Recognition** and **Image-Analysis services** are the primary candidates for scaling due to their computational intensity. As the number of cameras and image streams increases, these services will require additional resources to maintain performance levels. Once, the camera count grows, the Collector service must efficiently manage multiple streams of data. This can be achieved by scaling the number of **Collector-Service instances**, thereby distributing the processing load across more replicas. If each instance of the Camera service sends its stream to the Collector service, the load balancer in front of the Collector service can distribute incoming requests across the available pods to ensure that no single pod becomes overwhelmed. **Improving Scaling:** To support better scaling, we might consider implementing a queue-based system using a message broker. This would decouple the **Camera-Services** from the processing services. The Image-Analysis and Face-Recognition services could then process messages from the queue at their own pace. This also makes the system more fault-tolerant, as messages can be retained in the queue if a processing service goes down. A potential, scaling could be stateful services that require persistent data. If the Section-Service, for example, relies on a local database, it could become a bottleneck when scaling because each instance would need to handle its own data consistency and redundancy. Moving to a shared, scalable database service could reduce this issue.

## 7. Ingress Configuration

The Ingress configuration in cc-cluster (the assignment cluster) acts as a controller for routing external HTTP(S) traffic to the appropriate services within the Kubernetes cluster (using rewrite annotation).

- Ingress Controller: the Nginx Ingress Controller is used, which is responsible for handling and routing external traffic to the right services based on the defined rules.
- Annotations: The nginx.ingress.kubernetes.io/rewrite-target annotation is used to rewrite the URL path before forwarding the request to the appropriate service.
- Rules: Each rule maps a path to a service and its corresponding port. For instance, requests to 34.118.46.46.nip.io/alert-service are routed to the alert-service on port 8080. The same pattern is applied for other services. **Monitoring Tools:** Also, I used to **Forward ports** configured for **Prometheus** and **Grafana**, allowing external access to these monitoring tools.

**GKE Cluster Configuration:**

- Machine Type: E2 customize (4 RAM, 4 vCPUs).
- Autoscaling: Enabled to ensure that the cluster can automatically adjust the number of nodes as workloads demand.
- **Monitoring:** Activated monitoring features to monitor, log, and create performance metrics, and API.

## 8. Cost Analysis

Yearly Cost of a GKE Cluster: To estimate the yearly cost of running a GKE cluster, we must consider various components such as compute instances, persistent storage, network usage, load balancing services, and any other additional GCP services utilized. **For a basic cluster with E2 instances (4 vCPUs, 4GB RAM),** a rough estimate would be:

- Compute Costs: The E2 standard instance, costs approximately $0.14 per hour for on-demand instances. For one instance running continuously for a year, the compute cost would be about $0.14 * 24 * 365 = $1,226.4.
- Persistent Disk: Assume each node has a 100GB standard persistent disk. At approximately $0.040 per GB per month, this is $4 per month or $48 per year, per disk.

For a cluster with a single node, the basic annual cost would be around $1,226 (compute cost) + $48 (disk) = ~ $1,275. So, for 4 nodes (~ $5100).

- Autopilot GKE Cluster: The autopilot pricing model charges for the resources used by the workload pods. This means paying for the vCPU, memory, and storage resources that your pods request. Let's say each pod requests 1 vCPU and 2 GB of RAM, with a total of 16 pods to match the 4-node standard cluster vCPU resources. Autopilot charges $0.0446 per vCPU hour and $0.0055 per GiB hour. This equates to an annual cost of 16 * (0.0446 * 24 * 365) for vCPU and 16 * (0.0055 * 24 * 365 * 2) for memory.
- Google Cloud Run: To estimate the costs for Google Cloud Run, we need to know the number of requests and the execution time per request. Assuming the costs are to be matched with the GKE scenarios, we calculate as follows: 1 CPU and 512MB of memory per container instance. Cloud Run charges for CPU and memory allocated during request processing. The bills in 100ms increments, so with a 500ms execution time per request, we need to calculate the cost per 100ms and multiply by 5 for the execution time. Keeping 1 instance "warm" means we are paying for continuous usage.

The price differences between these services will primarily reflect the management overhead and scaling capabilities.

Buying and maintaining hardware includes servers, networking, storage, space, power, cooling, and expert staff. For a server that could handle similar workloads, it might cost several thousand dollars upfront. Plus, operational costs including electricity, cooling, and maintenance could easily add hundreds or more to the annual costs. For cloud vs on-premises hardware, **cloud** services offer significant advantages in terms of scalability, global distribution, and reduction in management overhead. For predictable, consistent workloads, **on-premises hardware** can sometimes be more cost-effective.

**Serverless:** Running serverless architectures, where the pay per invocation, can be more cost-effective for workloads. Google's Cloud Run is an example.

**Containerless:** is typically associated with PaaS (Platform as a Service). This can lead to cost savings due to the reduced need for infrastructure management and optimization.

**Clusters in GKE:** provides a more controlled environment for container orchestration with Kubernetes, which can be more cost-effective for larger or more complex applications.

## 9. Scaling Strategy on Kubernetes

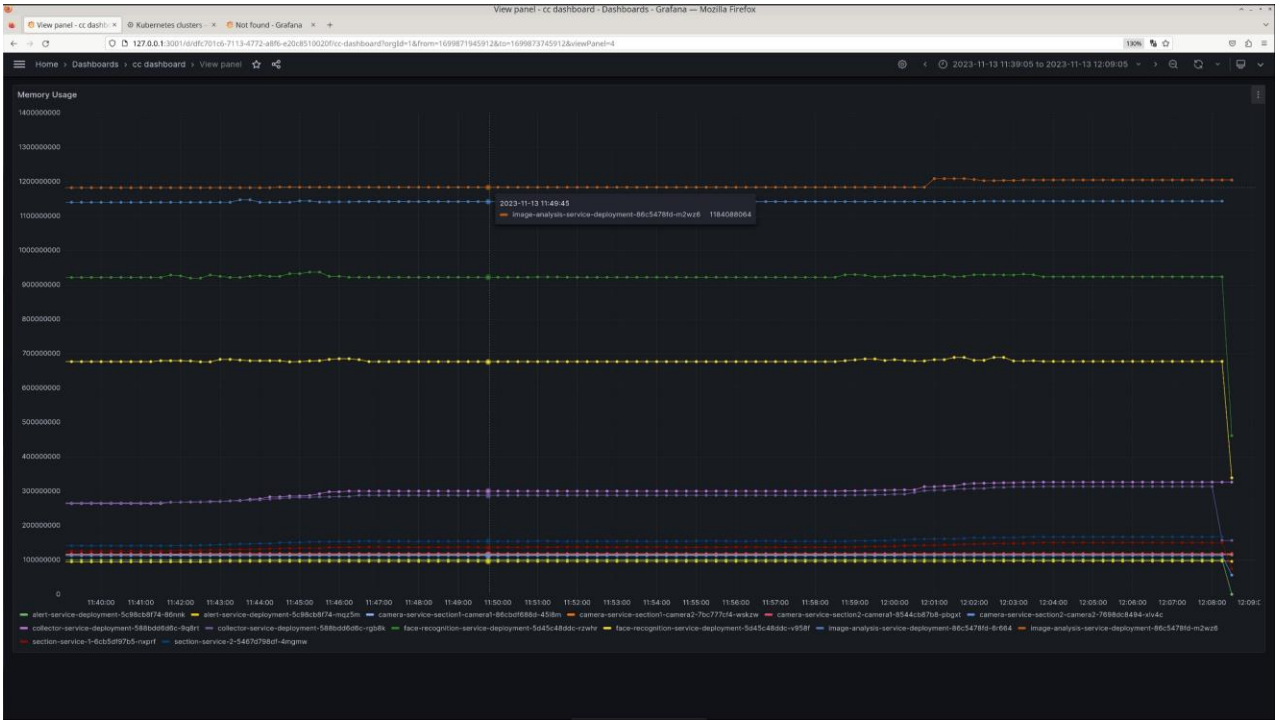
- Manual Scaling: I set an initial number of replicas for each service (2 replicas).
- Automatic Scaling with HPA: It automatically adjusts the number of running Pods in response to the CPU utilization:

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: section-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: section-service-deployment
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 80
```

In this configuration, I specified that the section-service should have a minimum of (2 Pod replicas), and can scale up to (10 replicas) if the CPU utilization hits 80%. This allows the service to handle increased load without my intervention.

- Usage of Cloud Run: Some services, particularly stateless ones that do not require persistent storage, could potentially be moved to Cloud Run. This could be beneficial for services that experience variable traffic and do not need the full capabilities of Kubernetes.
- Resource Usage: Each Pod has been allocated with 4 vCPUs and 4 GB of memory. With the deployed of HPA, with a minimum of 2 replicas, baseline resource allocation for the section-service would be at least 8 vCPUs and 8 GB of memory. As traffic increases, if the HPA scales the service up to 10 replicas, this could be consuming up to 40 vCPUs and 40 GB of memory. Monitoring tools like Prometheus and Grafana are instrumental in providing the insights needed to make informed decisions regarding scaling and resource allocation. They will track the resource usage over time and provide data to compare against my resource allocation to determine if my estimations were accurate or need adjustment.

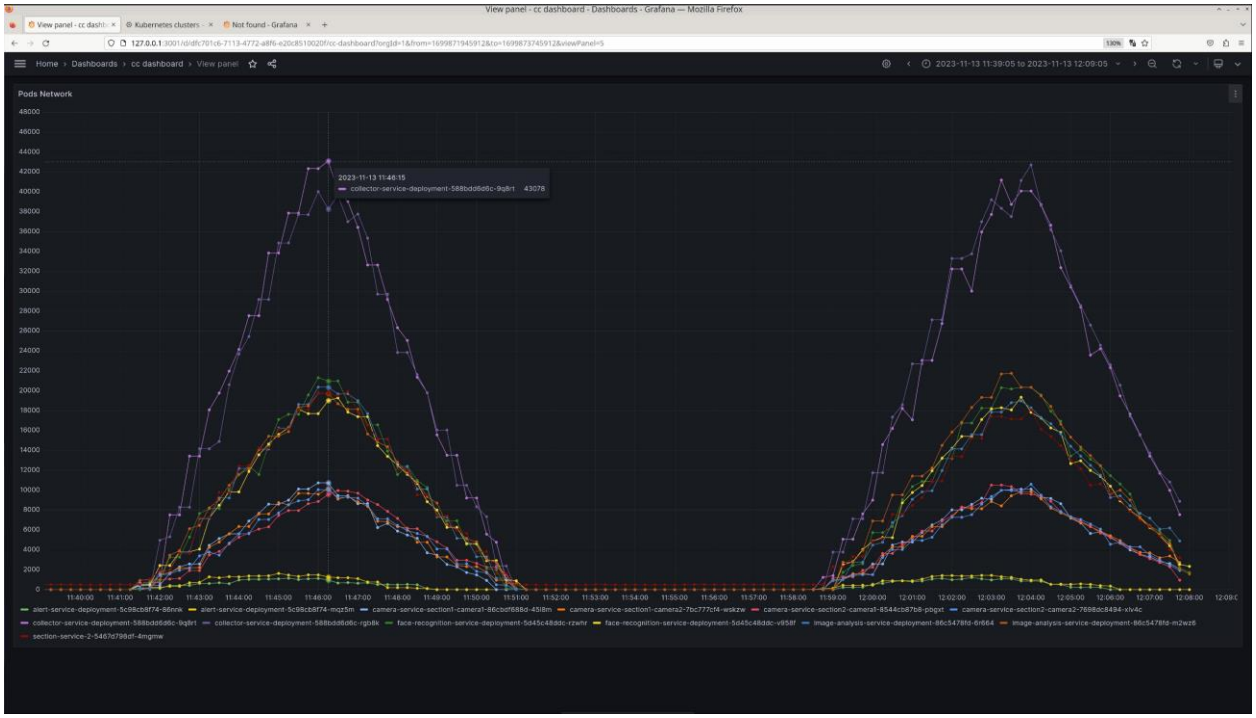
# 10. Screenshots



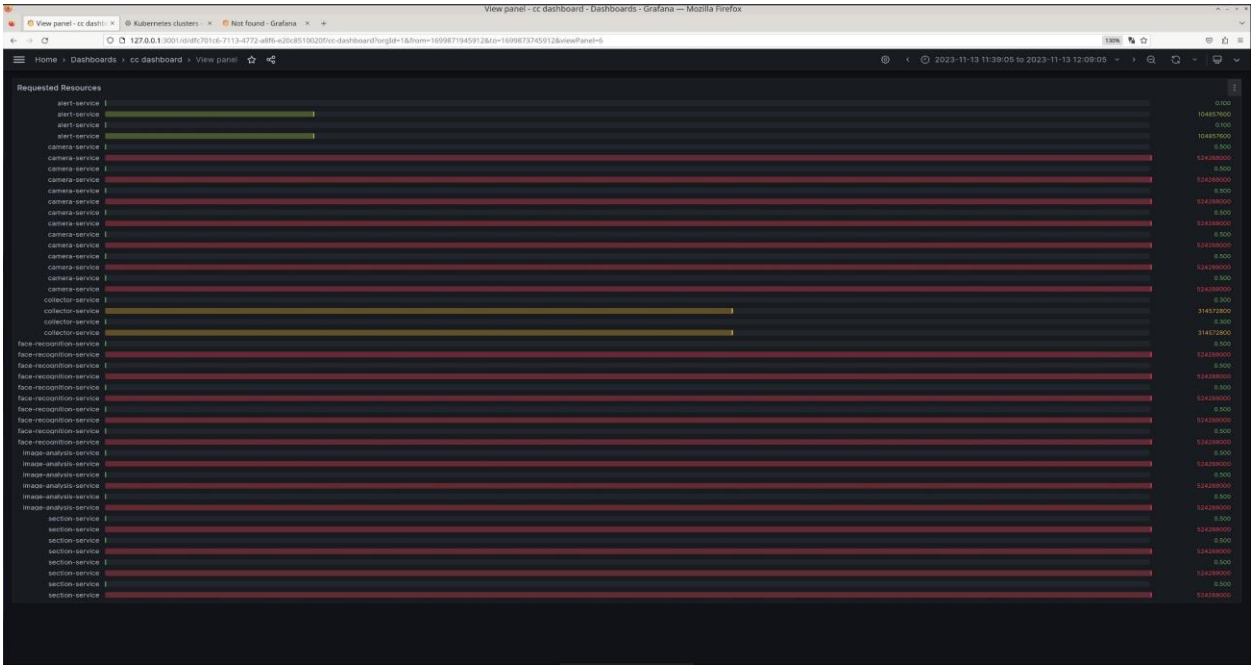

10.1 Pods Memory Usage


10.2 Pods CPU Usage

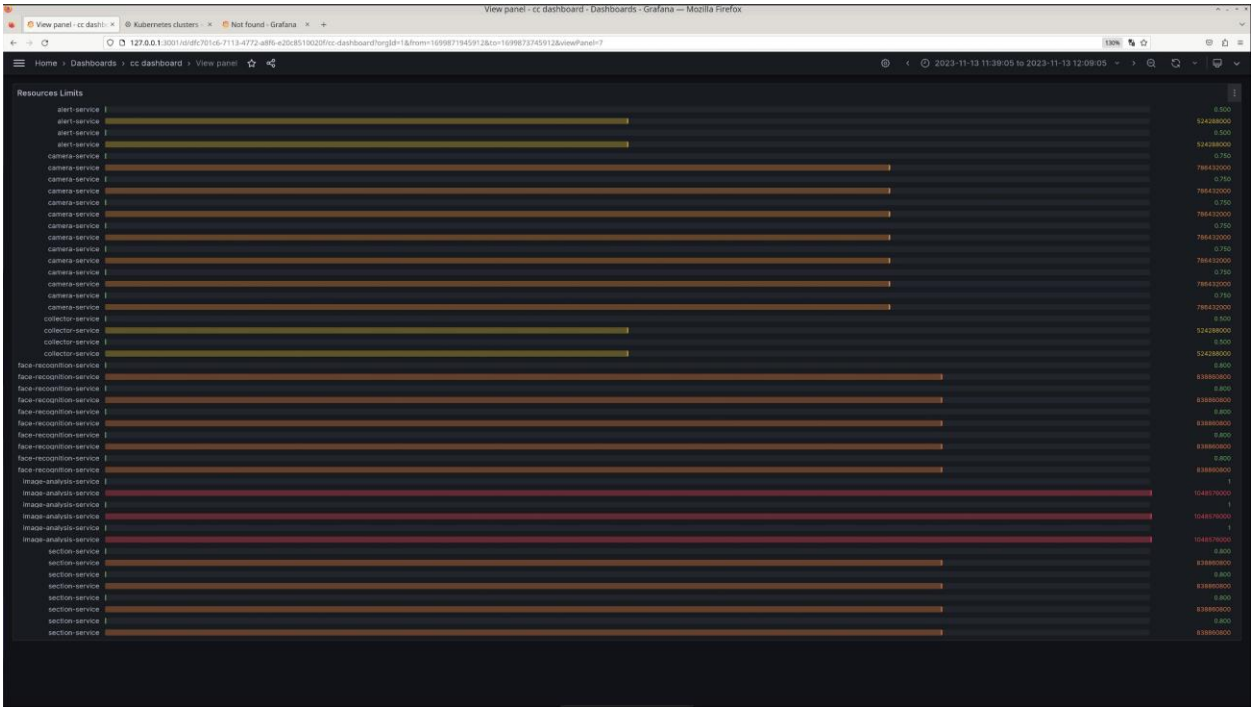

10.3 Pods Running Status



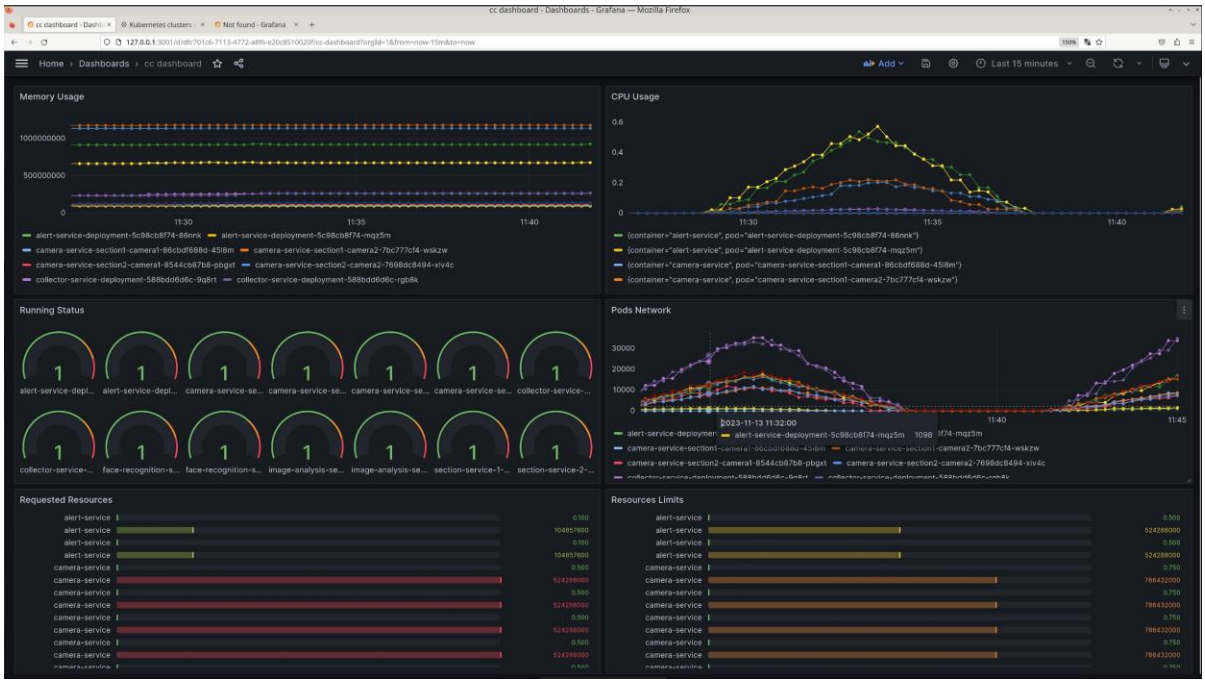

10.4 Pods Network Traffics




10.5 Service Requested Resources




10.6 Service Limited Resources




10.7 Grafana Dashboard

```
maith@debian: ~
File  Edit  View  Search  Terminal  Help
maith@debian:~$ kubectl get svc
NAME                                          TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)                      AGE
alert-service                                 ClusterIP      10.48.3.197     <none>          8080/TCP                     151m
alertmanager-operated                         ClusterIP      None            <none>          9093/TCP,9094/TCP,9094/UDP   134m
camera-service-section1-camera1               ClusterIP      10.48.9.89      <none>          8080/TCP                     151m
camera-service-section1-camera2               ClusterIP      10.48.5.122     <none>          8080/TCP                     151m
camera-service-section2-camera1               ClusterIP      10.48.6.246     <none>          8080/TCP                     151m
camera-service-section2-camera2               ClusterIP      10.48.15.91     <none>          8080/TCP                     151m
collector-service                             ClusterIP      10.48.11.145    <none>          8080/TCP                     151m
face-recognition-service                      ClusterIP      10.48.9.45      <none>          8080/TCP                     151m
grafana                                       ClusterIP      10.48.12.251    <none>          80/TCP                       3h29m
image-analysis-service                        ClusterIP      10.48.15.105    <none>          8080/TCP                     151m
kubernetes                                    ClusterIP      10.48.0.1       <none>          443/TCP                      3h36m
my-release-ingress-nginx-controller           LoadBalancer   10.48.12.121    34.116.141.72   80:32222/TCP,443:32274/TCP   3h31m
my-release-ingress-nginx-controller-admission ClusterIP      10.48.13.58     <none>          443/TCP                      3h31m
prometheus-grafana                            ClusterIP      10.48.7.27      <none>          80/TCP                       134m
prometheus-kube-prometheus-alertmanager       ClusterIP      10.48.2.193     <none>          9093/TCP,8080/TCP            134m
prometheus-kube-prometheus-operator           ClusterIP      10.48.9.107     <none>          443/TCP                      134m
prometheus-kube-prometheus-prometheus         ClusterIP      10.48.2.203     <none>          9090/TCP,8080/TCP            134m
prometheus-kube-state-metrics                 ClusterIP      10.48.7.223     <none>          8080/TCP                     134m
prometheus-operated                           ClusterIP      None            <none>          9090/TCP                     134m
prometheus-prometheus-node-exporter           ClusterIP      10.48.9.172     <none>          9100/TCP                     134m
section-service-1                             ClusterIP      10.48.3.244     <none>          8080/TCP                     151m
section-service-2                             ClusterIP      10.48.12.51     <none>          8080/TCP                     151m
sector-service-1                              LoadBalancer   10.48.13.77     34.118.40.38    8080:32570/TCP               3h1m
sector-service-2                              LoadBalancer   10.48.5.47      34.116.252.134  8080:31724/TCP               3h1m
maith@debian:~$
```

10.8 Kubernetes Services

```
File  Edit  View  Search  Terminal  Help
maith@debian:~$ kubectl get pods
NAME                                                READY   STATUS    RESTARTS   AGE
alert-service-deployment-5c98cb8f74-86nnk           1/1     Running   0          151m
alert-service-deployment-5c98cb8f74-mqz5m           1/1     Running   0          151m
alertmanager-prometheus-kube-prometheus-alertmanager-0   2/2     Running   0          135m
camera-service-section1-camera1-86cbdf688d-4518m    1/1     Running   0          151m
camera-service-section1-camera1-86cbdf688d-8c8j2    0/1     Pending   0          138m
camera-service-section1-camera2-7bc777cf4-wskzw     1/1     Running   0          151m
camera-service-section2-camera1-8544cb87b8-cznwl    0/1     Pending   0          140m
camera-service-section2-camera1-8544cb87b8-pbgxt    1/1     Running   0          151m
camera-service-section2-camera2-7698dc8494-l5bl4    0/1     Pending   0          138m
camera-service-section2-camera2-7698dc8494-xlv4c    1/1     Running   0          151m
collector-service-deployment-588bdd6d6c-9q8rt       1/1     Running   0          151m
collector-service-deployment-588bdd6d6c-rgb8k       1/1     Running   0          151m
face-recognition-service-deployment-5d45c48ddc-rzwhr   1/1     Running   0          151m
face-recognition-service-deployment-5d45c48ddc-v958f   1/1     Running   0          151m
grafana-7dc45955d7-zjchn                            1/1     Running   0          164m
image-analysis-service-deployment-86c5478fd-6r664   1/1     Running   0          151m
image-analysis-service-deployment-86c5478fd-m2wz6   1/1     Running   0          151m
my-release-ingress-nginx-controller-b5bcd47f5-j2k55   1/1     Running   0          164m
prometheus-grafana-6fdbffb5c9-dpqmg                 3/3     Running   0          135m
prometheus-kube-prometheus-operator-85cb49565b-zkvfz   1/1     Running   0          135m
prometheus-kube-state-metrics-898dd9b88-8qsgl       1/1     Running   0          135m
prometheus-prometheus-kube-prometheus-prometheus-0   2/2     Running   0          135m
prometheus-prometheus-node-exporter-dvtwc           1/1     Running   0          135m
prometheus-prometheus-node-exporter-n958z           1/1     Running   0          135m
prometheus-prometheus-node-exporter-vssg4           1/1     Running   0          135m
section-service-1-6cb5df97b5-jwr4t                  0/1     Pending   0          138m
section-service-1-6cb5df97b5-nxprf                  1/1     Running   0          151m
section-service-2-5467d798df-4mgmw                  1/1     Running   0          151m
section-service-2-5467d798df-bmzlx                  0/1     Pending   0          137m
maith@debian:~$
```

10.9 Kubernetes Pods

```
maith@debian: ~
File  Edit  View  Search  Terminal  Help
camera12-service-hpa          Deployment/camera21-service-section1-camera2   <unknown>/80%   2   10   0   153m
camera21-service-hpa          Deployment/camera-service-section2-camera1     0%/80%          2   10   2   153m
camera22-service-hpa          Deployment/camera-service-section2-camera2     0%/80%          2   10   2   153m
collector-service-hpa         Deployment/collector-service-deployment        0%/80%          2   10   2   153m
face-recognition-service-hpa  Deployment/face-recognition-service-deployment  27%/80%         2   10   2   153m
image-analysis-service-hpa    Deployment/image-analysis-service-deployment   9%/80%          2   10   2   153m
section1-service-hpa          Deployment/section-service-1                   1%/80%          2   10   2   153m
section2-service-hpa          Deployment/section-service-2                   1%/80%          2   10   2   153m
maith@debian:~$ kubectl get hpa
NAME                          REFERENCE                                      TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
alert-service-hpa             Deployment/alert-service-deployment            2%/80%          2         10        2          153m
camera11-service-hpa          Deployment/camera-service-section1-camera1     0%/80%          2         10        2          153m
camera12-service-hpa          Deployment/camera21-service-section1-camera2   <unknown>/80%   2         10        0          153m
camera21-service-hpa          Deployment/camera-service-section2-camera1     0%/80%          2         10        2          153m
camera22-service-hpa          Deployment/camera-service-section2-camera2     0%/80%          2         10        2          153m
collector-service-hpa         Deployment/collector-service-deployment        0%/80%          2         10        2          153m
face-recognition-service-hpa  Deployment/face-recognition-service-deployment  113%/80%        2         10        2          153m
image-analysis-service-hpa    Deployment/image-analysis-service-deployment   9%/80%          2         10        2          153m
section1-service-hpa          Deployment/section-service-1                   1%/80%          2         10        2          153m
section2-service-hpa          Deployment/section-service-2                   1%/80%          2         10        2          153m
maith@debian:~$ kubectl get hpa
NAME                          REFERENCE                                      TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
alert-service-hpa             Deployment/alert-service-deployment            2%/80%          2         10        2          154m
camera11-service-hpa          Deployment/camera-service-section1-camera1     1%/80%          2         10        2          154m
camera12-service-hpa          Deployment/camera21-service-section1-camera2   <unknown>/80%   2         10        0          154m
camera21-service-hpa          Deployment/camera-service-section2-camera1     0%/80%          2         10        2          154m
camera22-service-hpa          Deployment/camera-service-section2-camera2     1%/80%          2         10        2          154m
collector-service-hpa         Deployment/collector-service-deployment        11%/80%         2         10        2          154m
face-recognition-service-hpa  Deployment/face-recognition-service-deployment  139%/80%        2         10        3          154m
image-analysis-service-hpa    Deployment/image-analysis-service-deployment   49%/80%         2         10        2          154m
section1-service-hpa          Deployment/section-service-1                   1%/80%          2         10        2          154m
section2-service-hpa          Deployment/section-service-2                   1%/80%          2         10        2          154m
maith@debian:~$ kubectl get hpa
NAME                          REFERENCE                                      TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
alert-service-hpa             Deployment/alert-service-deployment            2%/80%          2         10        2          154m
camera11-service-hpa          Deployment/camera-service-section1-camera1     1%/80%          2         10        2          154m
camera12-service-hpa          Deployment/camera21-service-section1-camera2   <unknown>/80%   2         10        0          154m
camera21-service-hpa          Deployment/camera-service-section2-camera1     0%/80%          2         10        2          154m
camera22-service-hpa          Deployment/camera-service-section2-camera2     1%/80%          2         10        2          154m
collector-service-hpa         Deployment/collector-service-deployment        11%/80%         2         10        2          154m
face-recognition-service-hpa  Deployment/face-recognition-service-deployment  139%/80%        2         10        4          154m
image-analysis-service-hpa    Deployment/image-analysis-service-deployment   49%/80%         2         10        2          154m
section1-service-hpa          Deployment/section-service-1                   1%/80%          2         10        2          154m
section2-service-hpa          Deployment/section-service-2                   1%/80%          2         10        2          154m
maith@debian:~$
```

10.10 Kubernetes HPA