Collector

Since all other services, including the camera, are synchronous, I designed my solution with the thought that we need to free up the camera as soon as possible, and then we can continue processing the frames asynchronously (with regard to the cameras).

The processing of the frames clearly takes 2 flows:

1) The frame should be processed in Image Analysis, and then its result (if such is returned) is passed to the Section.
2) The frame should be processed through Face Recognition, and then its result (if such is returned) is passed to the Alert.

And those paths are independent, thus can be parallelized (processed asynchronously with regard to each other).

Therefore, the collector clearly needs to exhibit some asynchronous behavior, so I decided to develop it using the Python framework FastAPI, as it allows the creation of background processes (or green threads).

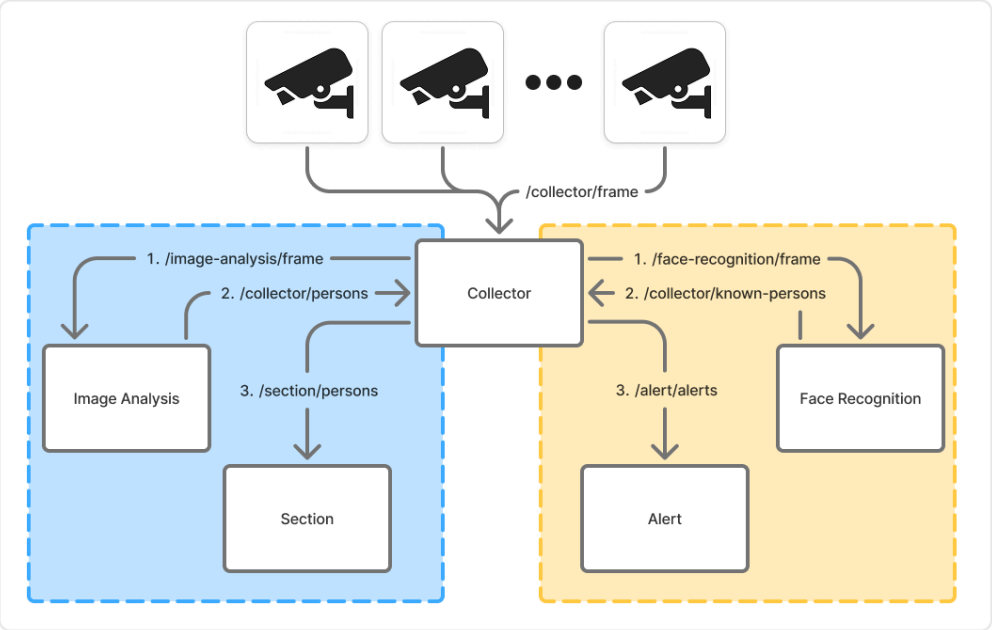
My initial approach was to spawn two background processes, following the flows described above, whenever the collector receives a frame, and free up the camera immediately. However, with a high enough frame rate of the camera, it was quickly visible that Image Analysis and Face Recognition were overloaded and had no time to respond within the timeout range, or even crashed.

Image Analysis and Face Recognition are synchronous, so it seemed like the Collector should wait for the service to finish its job before initializing a new connection. I decided to use a lock to achieve this. But shortly after, I realized that it is not the lock that should be introduced, but services that need to be scaled (replicated) - so that the collector could split the load between the copies rather than flooding a single service.

In addition, I decided to increase the timeout of the collector HTTP client to 10 seconds. This may have helped a bit.

Furthermore, my initial plan was to wait for a reply from the Image Analysis and Face Recognition, and then forward the result to the Section and Alert, respectively. But if I don't receive the reply within the timeout, the information is lost. However, Image Analysis and Face Recognition could do relevant work processing the frame and just send it a bit later. This is where I decided to introduce two additional optional routes to the Collector - /persons and /known-persons - and when I send to Image Analysis and Face Recognition, I set those collector routes as the destination where to return the reply. This way, even if we don't get a reply in time, we will eventually receive it.

The communication within my system can be depicted in the diagram:

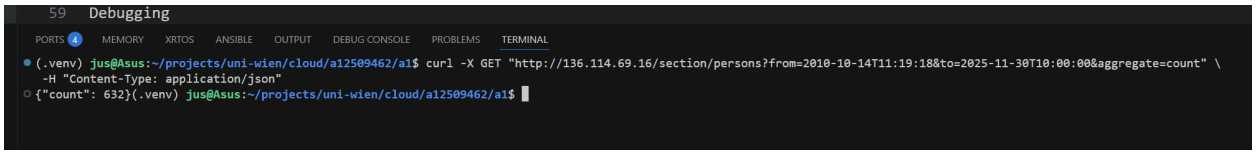


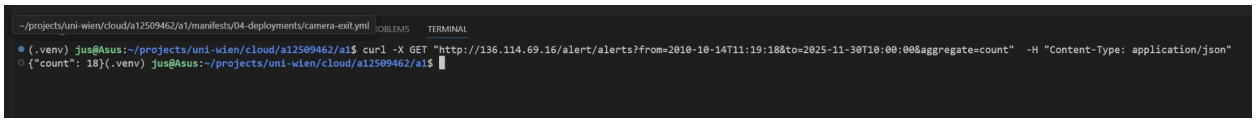
Frame flow:

1. The collector receives the frame from the camera. Replies immediately, and spawns two background processes (image_analysis and face_recognition) that forward the frame to Image Analysis and Face Recognition, respectively.
2. Two background processes:
    a. Image Analysis processes the frame and sends a reply with the destination attribute to the collector/persons
    b. Face Recognition processes the frame and sends a reply with the destination attribute to the collector/known-persons
3. Processing the replies:
    a. The Collector receives a reply from Image Analysis, and if the reply contains persons, it spawns a background process that forwards the result to the Section.
    b. The Collector receives a reply from Face Recognition, and if the reply contains known persons, it spawns a background process that forwards the result to the Alert.

This way:

- The Collector acts as an orchestrator: services only see the Collector, not each other.
- Frame processing is asynchronous and seems immediate to the Camera.
- The Image Analysis flow is asynchronous (executed in parallel) to the Face Recognition flow.
- The Section and Alert flow is independent of the Image Analysis and Face Recognition flows; it executes when a result is received from these two flows.
- Camera frames eventually reach the Section and Alert services


59 Debugging
PORTS 4 MEMORY XRTOS ANSIBLE OUTPUT DEBUG CONSOLE PROBLEMS TERMINAL
(.venv) jus@Asus:~/projects/uni-wien/cloud/a12509462/a1$ curl -X GET "http://136.114.69.16/section/persons?from=2010-10-14T11:19:18&to=2025-11-30T10:00:00&aggregate=count" \
-H "Content-Type: application/json"
{"count": 632}(.venv) jus@Asus:~/projects/uni-wien/cloud/a12509462/a1$


~/projects/uni-wien/cloud/a12509462/a1/manifests/04-deployments/camera-exit.yml
TERMINAL
(.venv) jus@Asus:~/projects/uni-wien/cloud/a12509462/a1$ curl -X GET "http://136.114.69.16/alert/alerts?from=2010-10-14T11:19:18&to=2025-11-30T10:00:00&aggregate=count" -H "Content-Type: application/json"
{"count": 18}(.venv) jus@Asus:~/projects/uni-wien/cloud/a12509462/a1$

Scalability and Bottlenecks.

During the tests on Kubernetes, it was clear that the heaviest services, which require a significant amount of resources and processing time, are Image Analysis and Face Recognition. In addition, since the entire system is highly dependent on the reliability of the Collector as the main orchestrator, it also needs to scale accordingly.

HPA

So I decided to introduce 3 HPAs - for the Collector, Image Analysis, and Face Recognition pods. Each with a minimum of 1 replica and a maximum of 10 replicas, and resource targets of CPU and memory with an average utilization of 80%. Whenever the average CPU or memory utilization across the pods reaches 80% of the requested resources, the Horizontal Pod Autoscaler triggers and begins scaling the deployment up by adding more replicas to distribute the load.

Resources Requests

First, I need to select the necessary amount of resources required for every service (not only the three that need to be scaled) on GKE.

To achieve this, I began with the amount that worked well on my local machine. Then I introduced a minimal load by starting a single camera with a delay of 1, which equals to frame rate of 1 frame per second (remember I free up the camera immediately). 1 fps is a good enough frame rate for security cameras; there is no need to stream at, say, 60 fps.

What I wanted to achieve now is that the system would work reliably and not spawn replicas for a single camera setup. However, if the load increases (more cameras are introduced), it should scale accordingly.

By examining collector logs (to confirm that frames are not lost), pod metrics using the kubectl top pods command, and Grafana panels, I observed peaks in CPU and memory utilization. Since I set the average utilization in HPA targets to 80% for both CPU and memory, and I don't want more replicas with a single camera, I set the resource requests for each service to PEAK/80% and rounded them up to the nearest ten.

Screenshot from the namespace resources panel - Grafana. I used it to go to each service resources panel and wrote down the peaks together with the values I have seen from the kubectl top pods command

This way I set resource requests to:

	Camera	Alert	Section	Collector	Image	Face	Mongo
CPU	10m	10m	10m	100m	550m	600m	10m
Memory	100Mi	80Mi	130Mi	200Mi	700Mi	650Mi	150Mi

Resources Limits

After the resource requests were set, I calculated the resource limits again by examining the spikes (and later tested, including the highest load of 4 cameras after boot streaming). I have made a good margin for services that I do not scale horizontally. Additionally, there is a good margin for the collector, as observed CPU throttling (seen on the Grafana panel). And a good margin for image analysis and face recognition to allow bursting to withstand spikes.

Limits:

	Camera	Alert	Section	Collector	Image	Face	Mongo
CPU	50m	100m	100m	350m	1000m	1000m	100m
Memory	125Mi	150Mi	175Mi	250Mi	850Mi	800Mi	200Mi

Increasing load

After I gathered the necessary resources for a single-camera setup (so that the entire system would withstand that load without replication), I began to increase the number of cameras.

I tested both a gradual increase of the load (jump from 1 camera streaming to 2 cameras streaming to 4 cameras streaming). And an immediate increase (4 cameras started streaming at the same time) - the latter is more important.

Even though I also tested the system with up to 2 entry and 2 exit cameras, the following experiments were conducted with a single camera with an increasing streaming rate.

Streaming Rate of a single camera	1	0.5	0.25
Equivalent Number of Cameras streaming at 1 fps each	1	2	4

It differed from run to run (for 2 and 4 cameras), but here are the HPA metrics for 1,2,4 cameras streaming respectively:

1 camera:

```
Every 1.0s: kubectl get hpa -n a1-app

NAME                  REFERENCE                     TARGETS                          MINPODS   MAXPODS   REPLICAS   AGE
collector             Deployment/collector          cpu: 36%/80%, memory: 77%/80%   1         10        1          7m38s
face-recognition-hpa  Deployment/face-recognition   cpu: 60%/80%, memory: 71%/80%   1         10        1          7m37s
image-analysis-hpa    Deployment/image-analysis     cpu: 60%/80%, memory: 74%/80%   1         10        1          7m37s
```

2 cameras equivalent:

```
Every 1.0s: kubectl get hpa -n a1-app

NAME                  REFERENCE                     TARGETS                          MINPODS   MAXPODS   REPLICAS   AGE
collector             Deployment/collector          cpu: 116%/80%, memory: 77%/80%   1         10        2          8m43s
face-recognition-hpa  Deployment/face-recognition   cpu: 143%/80%, memory: 77%/80%   1         10        2          8m42s
image-analysis-hpa    Deployment/image-analysis     cpu: 95%/80%, memory: 74%/80%    1         10        2          8m42s
```

Here, I captured the point at which it created a second replica for face recognition, image analysis, and the collector.
4 cameras equivalent:

```
NAME                  REFERENCE                     TARGETS                          MINPODS   MAXPODS   REPLICAS   AGE
collector             Deployment/collector          cpu: 110%/80%, memory: 78%/80%   1         10        3          10m
face-recognition-hpa  Deployment/face-recognition   cpu: 126%/80%, memory: 82%/80%   1         10        4          10m
image-analysis-hpa    Deployment/image-analysis     cpu: 48%/80%, memory: 77%/80%    1         10        3          10m
```

Sometimes it created fewer replicas, sometimes more, but it was always between 2-4 replicas.

I also needed to decrease the stabilization window in HPAs to 14 seconds for scale-up, so it could react quickly based on a single metric resolution (which, in my case, was 15 seconds). Similarly to 30 seconds for scale-down, to see the decrease in replicas when the load is down. In addition, I needed to increase timeoutSeconds to 10 and periodSeconds to 15 for the probes of those three services, because when flooding them with too many requests, they had no time to reply to the probes, and Kubernetes killed them, even though they were technically still healthy but busy with other tasks.

If I were to introduce a load of 4 cameras streaming immediately after the pods boot up, I could have seen image analysis and face recognition error for a couple of seconds (probably services were overloaded), and then image recognition scaled to 3 and face recognition scaled to 3, and recovery was quite fast (<30 sec). After the scale system continued processing the frames, so became healthy again.

On the contrary, with a gradual increase in the load from 1 to 2 to 4 cameras, at the point where 4 cameras started streaming, there were already two replicas of each of the three services (replication happened due to the 2-camera load). Thus, I have seen no problems in the collector logs.

Apart from that, there were no problems in the collector logs, so the system handled the increase in load perfectly in other cases. From the very beginning, I have also noticed that image recognition occasionally returns a 403 error on some frames, and retries with the same frame do not help - therefore, I drop those frames since they occur very rarely. Additionally, in the event that some services are unavailable, Kubernetes quickly recovers them.

If one were to introduce more sections, my logic is that the entire system is replicated. It makes sense to attach cameras to the particular section. Of course, one could reuse the same workers (collector, image analysis, face recognition). However, in my opinion, it also makes sense to deploy separate copies of them to keep them separate. Because, anyway, if more load is applied to workers, they will replicate.

Manifests, Kubernetes Objects, Ingress

Namespace: I created the namespace and then attach all my pods to it to organize them.

Deployment: For each app (wrapped in a Docker container), I introduced a Deployment object.

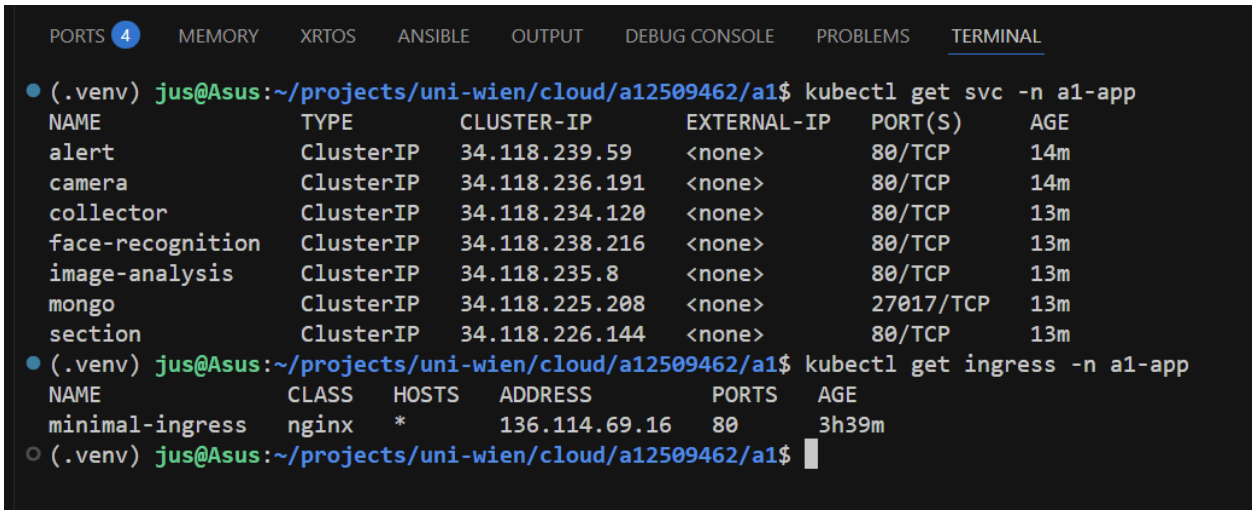
Service: For each deployment, I introduced a Service object with the type ClusterIP to enable local communication between services. For the collector, image analysis, and face recognition, the Service also serves as a load balancer.

Metrics, HPA: For HPA to work, I copied the manifest for the metrics server and introduced HPAs themselves for the collector, image analysis, and face recognition.

Volumes: To prevent the section state from being lost, since pods cannot be considered persistent, I introduced a MongoDB deployment and a persistent volume claim of 1 Gi for it. That turned out to be enough:

Ingress: For ingress to work, I copied the NGINX ingress controller manifest and introduced minimal ingress to allow external connections to the camera, section, and alert services.

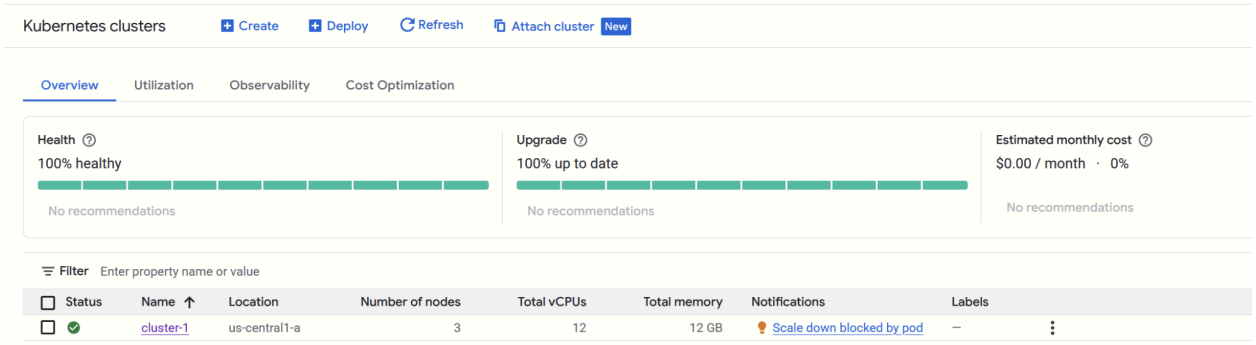
Also note that for my tests, I used a setup with a single camera and an increasing frame rate. If I were to introduce more cameras and wanted to make them start at the same time, I need a way to communicate with them, ideally not through the single load-balancing Service ClusterIP object (because it does not allow sending requests to each pod). One option would be to introduce a Service object for each Camera or apply tricks, such as https://www.baeldung.com/ops/kubernetes-cluster-service-requests-all-pods.



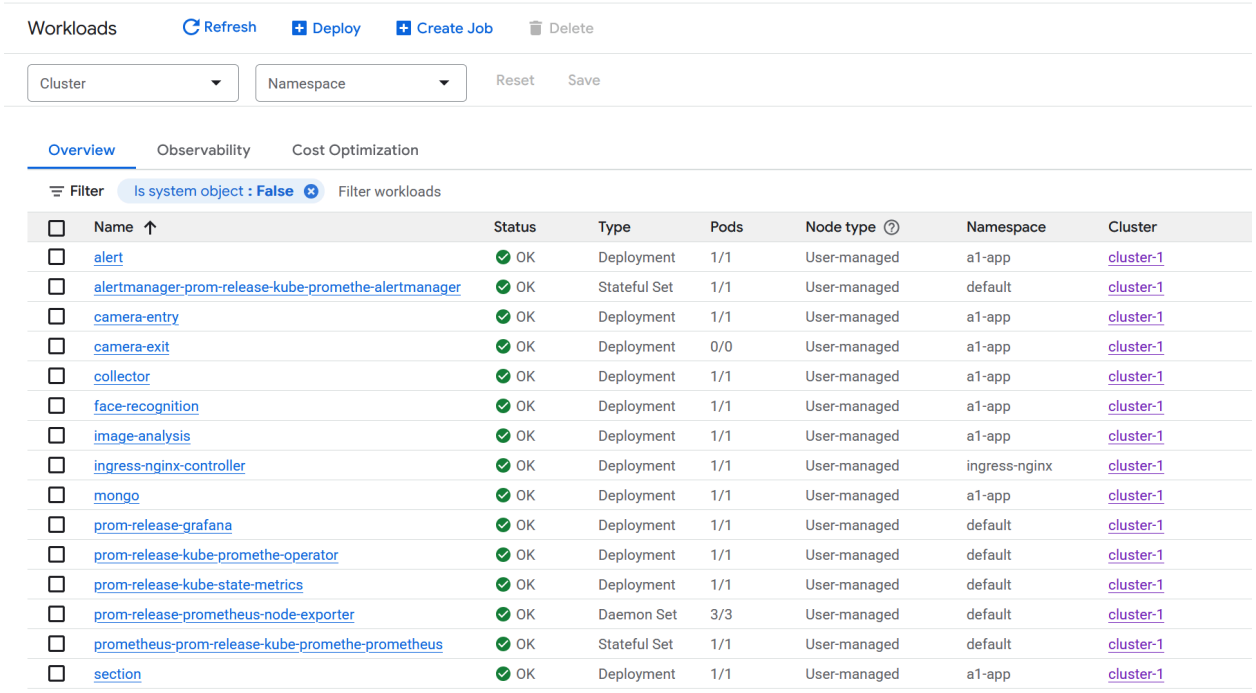
GKE cluster, Resource usage

I have followed the suggested one.

Initially, I selected a cluster with 6 GB of memory, but for some reason, Image Recognition won't boot up due to OOM. So I decided to go with the suggested upper bound, which worked fine:



Workloads



Workloads

Refresh Deploy Create Job Delete

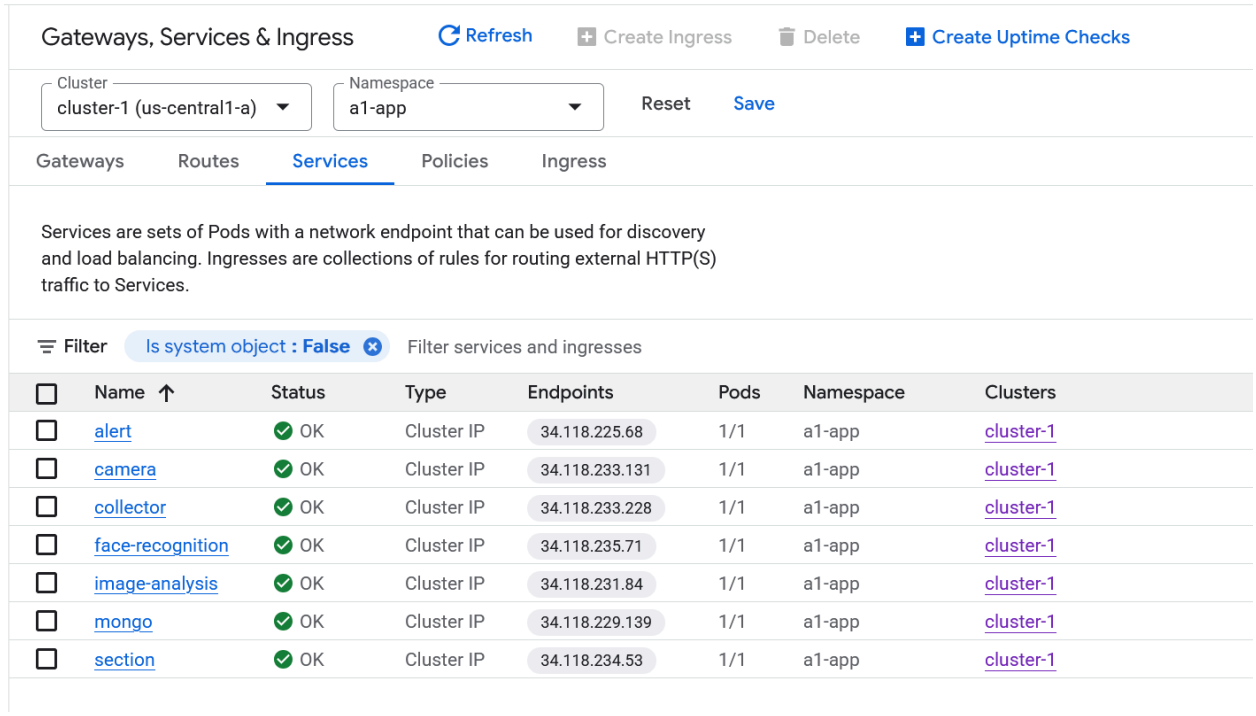
Cluster Namespace Reset Save

Overview Observability Cost Optimization

Filter Is system object : False Filter workloads

Name	Status	Type	Pods	Node type	Namespace	Cluster
alert	OK	Deployment	1/1	User-managed	a1-app	cluster-1
alertmanager-prom-release-kube-promethe-alertmanager	OK	Stateful Set	1/1	User-managed	default	cluster-1
camera-entry	OK	Deployment	1/1	User-managed	a1-app	cluster-1
camera-exit	OK	Deployment	0/0	User-managed	a1-app	cluster-1
collector	OK	Deployment	1/1	User-managed	a1-app	cluster-1
face-recognition	OK	Deployment	1/1	User-managed	a1-app	cluster-1
image-analysis	OK	Deployment	1/1	User-managed	a1-app	cluster-1
ingress-nginx-controller	OK	Deployment	1/1	User-managed	ingress-nginx	cluster-1
mongo	OK	Deployment	1/1	User-managed	a1-app	cluster-1
prom-release-grafana	OK	Deployment	1/1	User-managed	default	cluster-1
prom-release-kube-promethe-operator	OK	Deployment	1/1	User-managed	default	cluster-1
prom-release-kube-state-metrics	OK	Deployment	1/1	User-managed	default	cluster-1
prom-release-prometheus-node-exporter	OK	Daemon Set	3/3	User-managed	default	cluster-1
prometheus-prom-release-kube-promethe-prometheus	OK	Stateful Set	1/1	User-managed	default	cluster-1
section	OK	Deployment	1/1	User-managed	a1-app	cluster-1

Services



Gateways, Services & Ingress

Refresh Create Ingress Delete Create Uptime Checks

Cluster: cluster-1 (us-central1-a) Namespace: a1-app Reset Save

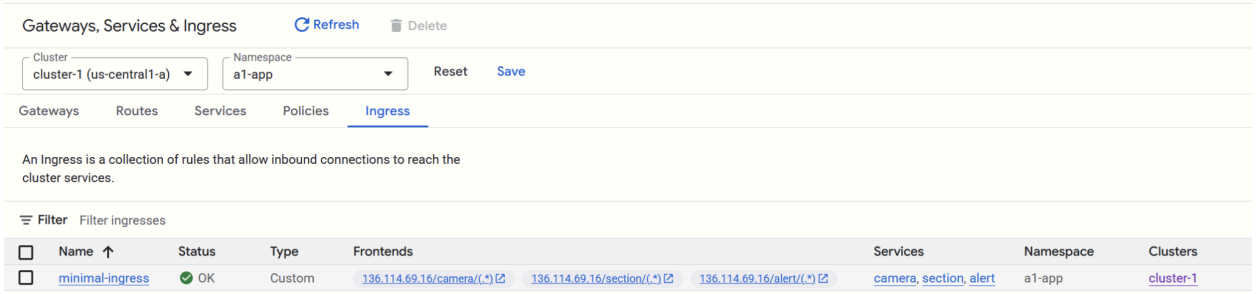
Gateways Routes Services Policies Ingress

Services are sets of Pods with a network endpoint that can be used for discovery and load balancing. Ingresses are collections of rules for routing external HTTP(S) traffic to Services.

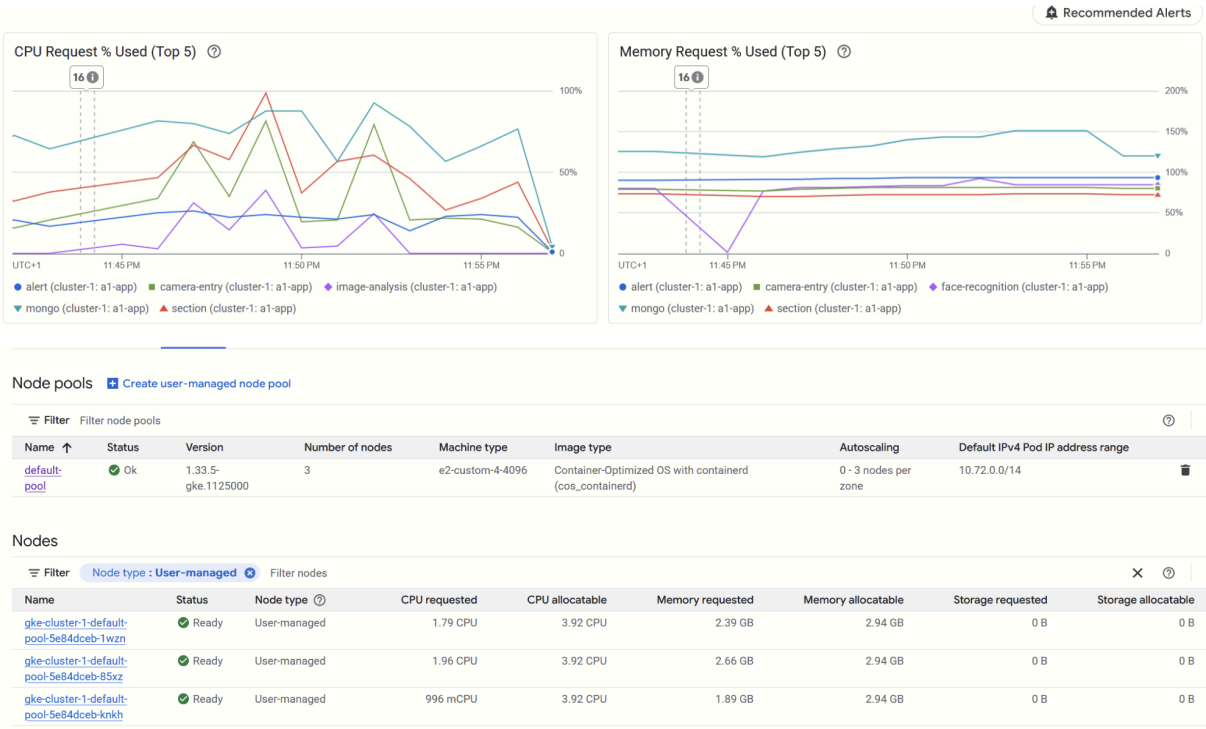
Filter Is system object : False Filter services and ingresses

Name	Status	Type	Endpoints	Pods	Namespace	Clusters
alert	OK	Cluster IP	34.118.225.68	1/1	a1-app	cluster-1
camera	OK	Cluster IP	34.118.233.131	1/1	a1-app	cluster-1
collector	OK	Cluster IP	34.118.233.228	1/1	a1-app	cluster-1
face-recognition	OK	Cluster IP	34.118.235.71	1/1	a1-app	cluster-1
image-analysis	OK	Cluster IP	34.118.231.84	1/1	a1-app	cluster-1
mongo	OK	Cluster IP	34.118.229.139	1/1	a1-app	cluster-1
section	OK	Cluster IP	34.118.234.53	1/1	a1-app	cluster-1

Ingress



Resource usage



Node pools

Create user-managed node pool

Filter
Filter node pools

Name	Status	Version	Number of nodes	Machine type	Image type	Autoscaling	Default IPv4 Pod IP address range
default-pool	Ok	1.33.5-gke.1125000	3	e2-custom-4-4096	Container-Optimized OS with containerd (cos_containerd)	0 - 3 nodes per zone	10.72.0.0/14

Nodes

Filter
Node type : User-managed
Filter nodes

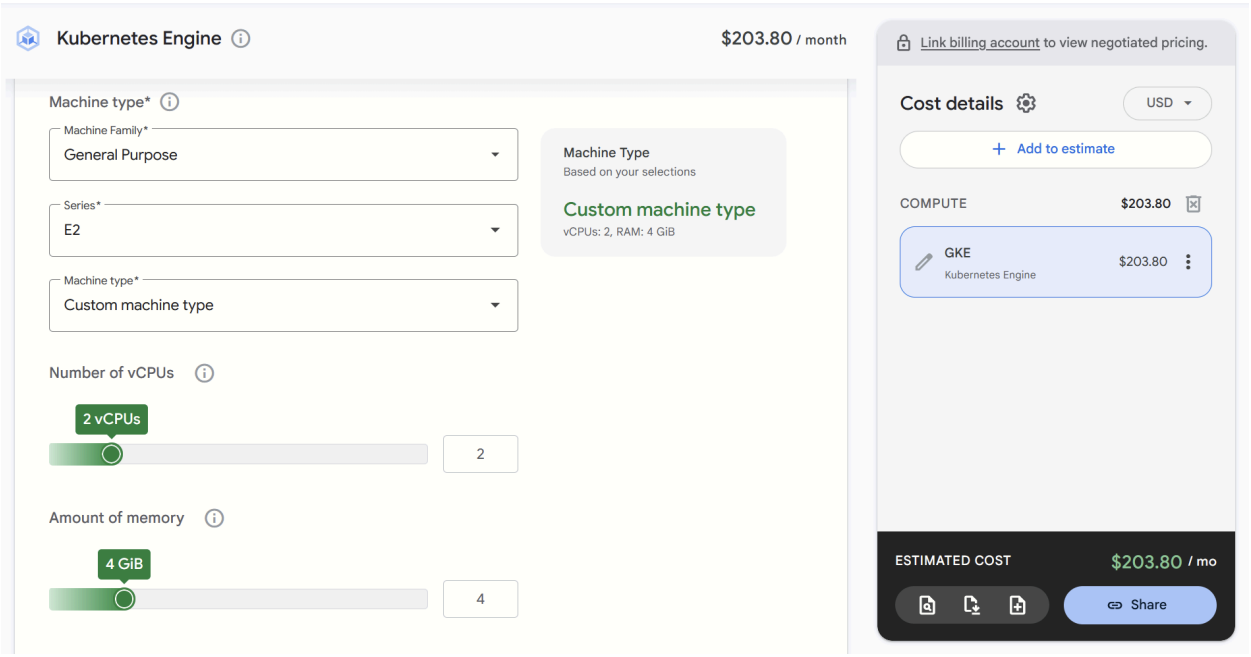
Name	Status	Node type	CPU requested	CPU allocatable	Memory requested	Memory allocatable	Storage requested	Storage allocatable
gke-cluster-1-default-pool-5e84dceb-1wzn	Ready	User-managed	1.79 CPU	3.92 CPU	2.39 GB	2.94 GB	0 B	0 B
gke-cluster-1-default-pool-5e84dceb-85xz	Ready	User-managed	1.96 CPU	3.92 CPU	2.66 GB	2.94 GB	0 B	0 B
gke-cluster-1-default-pool-5e84dceb-knkh	Ready	User-managed	996 mCPU	3.92 CPU	1.89 GB	2.94 GB	0 B	0 B

In fact, fewer resources were requested than allocated, so I could decrease the values (for CPU, in particular).

Cost Analysis

Standard GKE

To be able to withstand 4 cameras streaming, I could use fewer, 2 vCPU, for each node. This way, the price will be the following for a Standard GKE cluster:



Or about $2,445 yearly.

That's about $815 per node per year, which is actually a very efficient setup for small workloads.

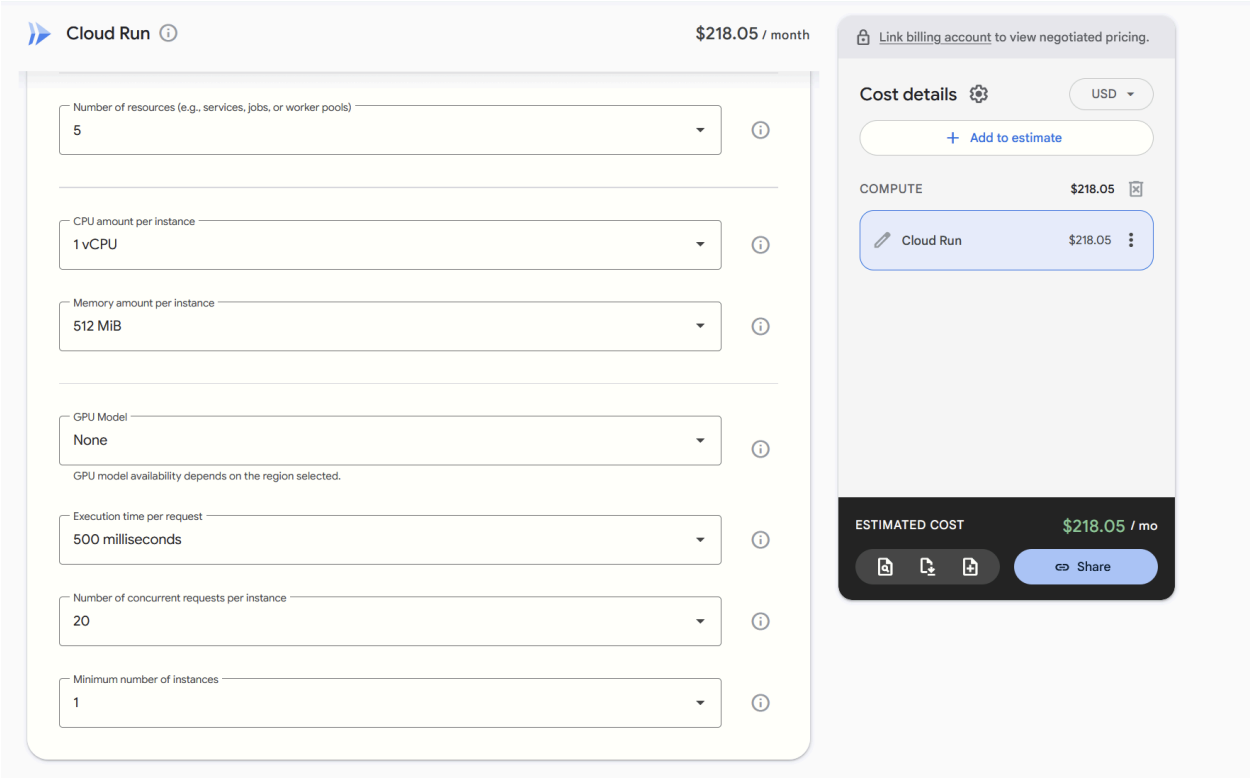
Component	On-Prem Estimate (/node)	Total (3 Nodes)
Hardware (2 vCPU, 4 GB RAM equivalent)	~$800 (one-time)	~$2,400
Power + Cooling (per year)	~$100/year → $300 (3 years)	~$900
Maintenance / Parts / Network (per year)	~$100/year → $300 (3 years)	~$900
3-Year Total (TCO)	$1,400 per node (800 + 300 + 300)	$4,200 total (3 nodes)

So roughly:

- **On-prem 3-year cost:** ≈ $4,000 - $5,000
- **Standard GKE 3-year cost:** $2,445.60 × 3 ≈ **$7,336.80**

So cloud costs 1.5x more, but eliminates nearly all management overhead and provides enterprise-grade uptime and scaling. Cloud is still a better choice to test the startup before investing big funds into hardware, and it scales well.

Cloud Run:
For 2 workers price is pretty similar /node as in GKE.



Cloud Run can handle 3M instead of 10M requests with roughly the same price.

GKE Standard: I pick node types. CPU/memory is dedicated per VM node; pods share it. Idle nodes still cost me.

GKE Autopilot: I pick pod resources; Google decides how many nodes to run. CPU/memory cost is per pod allocation (I don't see the node).

Cloud Run: CPU and memory are per container instance. Can scale to 0. "CPU always allocated" vs "CPU only during request" affects billing.

Cloud Vision alternatives

We can make use of Face Detection (https://cloud.google.com/vision/docs/detecting-faces) of Cloud Vision instead of Image Analysis. As for Face Recognition, unfortunately, Face Detection

does not support specific individual facial recognition. However, apparently, they have it in the pricing list, so let's rely on it.

Having 4 cameras streaming with 1 fps is about 10M images per month.

Facial Detection	Free	$1.50	$0.60
Facial Detection - Celebrity Recognition	Free	$1.50	$0.60

We need 2 units (one for Facial Detection - alternative for Image Analysis, and one for Celebrity Recognition - alternative for Face Recognition). That's $1.2 per 1000 images or $12,000 per month. So developing your own programs is beneficial.