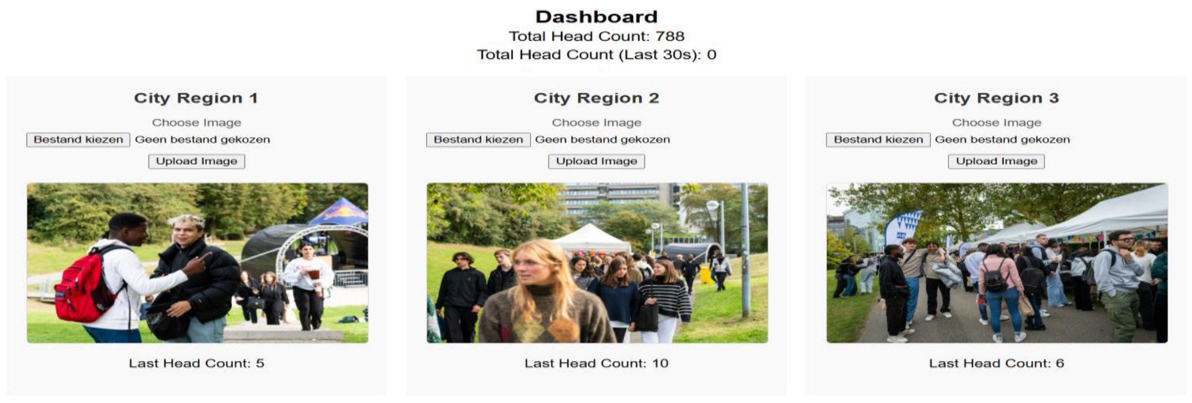
Cloud Computing

Alexander Dujardin (2nd years master) – 0592036

Functionalities

First I will go over the functionalities implemented for this application.


Dashboard
Total Head Count: 788
Total Head Count (Last 30s): 0

City Region 1
Choose Image
Bestand kiezen Geen bestand gekozen
Upload Image
Last Head Count: 5

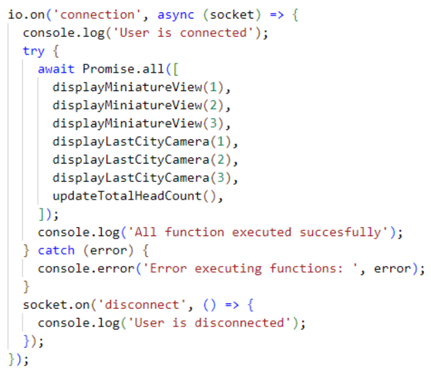
City Region 2
Choose Image
Bestand kiezen Geen bestand gekozen
Upload Image
Last Head Count: 10

City Region 3
Choose Image
Bestand kiezen Geen bestand gekozen
Upload Image
Last Head Count: 6

Frontend and backend:

1. City cameras, the system works with an array of “city cameras”, each described by a unique upload zone identifier. To simulate the process of capturing and uploading images from these camera, the user can upload images manually trough a form. These upload zones represents the different city regions, these uploaded images will then be processed to gain insights about head counts in the images.
   As seen in the above figure example of the user interface, we have three city regions, each having its own upload zone, every time an image is uploaded in one of these the image Is coupled to unique identifier 1, 2 or 3. These uploaded images in the frontend user interface are uploaded to the server, when the user clicks the button to upload, it sends a POST request with the form data to the endpoint ‘/upload/:uploadZone’. The backend handles the image uploads using Multer, which stores the images temporary in memory. These images are then send to the component where the headcount prediction software is running, this software will take an image, do some preprocessing techniques like edge detection, then change the image to a features set and run a neural network on it to predict the number of persons (head count) in the image. This head count result will be send back to the backend of the application, which will then also store the image with its upload zone identifier, datetime of creation, and head count to a MySQL database. With the resulted head count the server will do further updates to the user dashboard.

When a new client connects, it triggers a couple of functions: “UpdateTotalHeadCount”, “displayMinitatureView”, “displayLastCityCamera”: When a client connects, these asynchronous functions are executed to update the frontend with the latest data. These functions query the database for the most recent images and head counts, and then emit this data to the frontend using Socket.IO. This approach ensures that each client connecting receives the most current information.

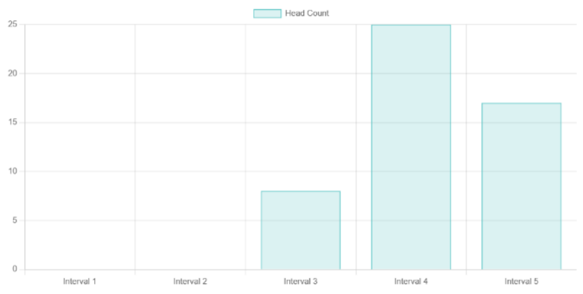


```
io.on('connection', async (socket) => {
  console.log('User is connected');
  try {
    await Promise.all([
      displayMiniatureView(1),
      displayMiniatureView(2),
      displayMiniatureView(3),
      displayLastCityCamera(1),
      displayLastCityCamera(2),
      displayLastCityCamera(3),
      updateTotalHeadCount(),
    ]);
    console.log('All function executed succesfully');
  } catch (error) {
    console.error('Error executing functions: ', error);
  }
  socket.on('disconnect', () => {
    console.log('User is disconnected');
  });
});
```

2. Total head count, in the top of the dashboard it says the total head count of all times of this application, each time a client connects to the server, the backend will retrieve the total head count from the MySQL database by taking the sum.
3. Camera specific head count, each upload zone has an own last head count, which says the head count of the last uploaded image for each camera, which is also the head count of the image showing in the miniature view.

The successful uploading of a new image not trigger these functions above but will trigger "imageUploaded", which updates the miniature view of that city camera to the new image, update the last head count to the head count of the just uploaded image, and update the total head count by summing up the total head count we already have and the new head count of the image. So when a new image is uploaded it doesn't need to retrieve the information again from the MySQL database, but directly use the processed information. The backend uses Socket.IO for real-time communication between the server and connected clients. By this it instantly updated the dashboards information, ensuring all clients receive latest information without refreshing the page.

4. Global head count dashboard, the user interface also shows a real-time dashboard where each bar represents an interval of 30 seconds, interval 1 starts when the client connects, and after every 30 seconds it is calculated what the total head count was of the uploaded images in this interval, so each 30 seconds the backend emits the new information to the frontend.

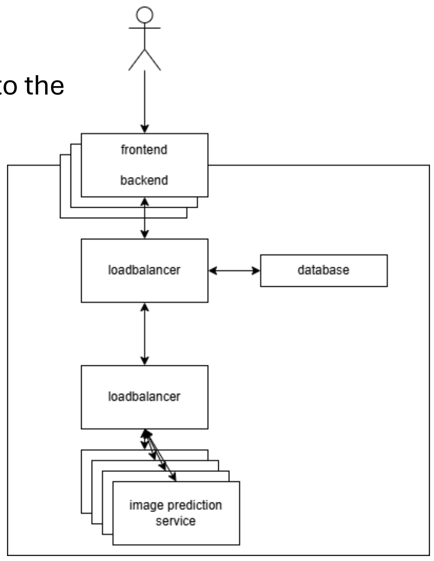


The working functionalities are shown in demo 1.

Architecture

For the deployment of the application I used Docker for containerization (for encapsulating all dependencies and libraries needed to run) and Kubernetes for orchestration (for automated scaling, deploying, operating, … from the applications containers). To containerize each component I created Docker files to make Docker images (immutable snapshots of the application environment). Then I created Kubernetes YAML files for defining the deployments (to manage scalability, creation, … of pods) and services (used to expose the application outside the cluster or to other components of the application) for the different components. This setup helps for managing the applications architecture, and gives features that are usable for fault tolerance, scalability and, efficient resource utilization. These are also the main non functional goals for the project which will be discussed later. For simulating the Kubernetes environment, I used Minikube which is a single node Kubernetes cluster, where this node hosts the application pods.

There are three main components:



1. **Frontend – backend**: I decided to combine the frontend and backend into the same component and work with plain html, css, and js because of the simplicity of the application but normally when designing more complex applications architecture it is better to separate these components and work with React.
   - Deployment ‘frontend-backend’ and service ‘front-backend-service’, which is of the type of LoadBalancer on port 3000, this to evenly distribute the incoming requests (for example when new users connect) across the available pods. On this deployment I will enable HPA, to automatically increase the number of pods of this deployment (this will be discussed in more detail in the next section).
2. **Image Prediction**: Head count prediction software (software as a service): Post the image on: http://image-predict-service:8002/crowdy/image/count and the response will be the head count of the image. The backend server in the frontend/backend component sends the images to this service via HTTP POST request using Axios (used for handling asynchronous HTTP requests). This service is responsible for processing uploaded images to predict head counts using a neural network.
   - Deployment ‘image-predict’ and ‘image-predict-service’, which is of the type of LoadBalancer on port 8002, this to evenly distritbute incoming image processing request across the available pods. On this deployment I will enable HPA, to automatically increase the number of pods of this deployment, because image processing can be CPU-intensive and workload-dependent. (I will in the next section discuss why I did this)
3. **MySQL component** serving as the primary data storage for the application, storing the images, meta-data and the head count. The backend uses the mysql2 package to interact with the database. It is not a single connection, but by a connection pool, which optimizes the performance by reusing database connections. This can handle multiple concurrent database operations (which is needed in a real time environment with the possibility of multiple clients). The database “images”, containing a table “images_table” with the database schema: id INT PRIMARY KEY AUTO_INCREMENT, image LONGTEXT NOT NULL, creation_date DATETIME NOT NULL, upload_zone INT NOT NULL, head_count INT NOT NULL The component consists of different Kubernetes objects that work together ensuring persistence, configuration and accessibility:
   - ‘mysql-configmap’: This is ConfigMap file which stores configuration data such as environment variables, including time zone settings. The time zone configuration is important because the application deals with timestamps, and synchronization between the database and the frontend-backend deploymen. (TZ: Europe/Brussels)
   - ‘mysql-pv’ and ‘mysql-pvc’: PersitVolume defines storage in the cluster that is provisioned for the MySQL database. I specified the capacity of 5Gi. PersistentColumeClaim: this claim requests the defined storage from the PersistentVolume. The use of PersistentVolume and PersistentVolumeClaim ensures that the database data is stored persistently and remains intact even if the MySQL pod is restarted or rescheduled (on a different node).

- ‘mysql-deployment’: here we specify the Docker image (‘mysql:latest’), environment variables such as TZ from configmap, database name, mysql root password, volume mounts, … .
- ‘mysql-service’: This service exposes the MySQL deployment within the Kubernetes cluster, allowing other components internal access on port 3306.

Some key design considerations that were made:

The asynchronous nature of the backend ensures that tasks such as database queries and data fetching do not block the main execution thread, allowing the server to handle multiple client connections efficiently. The use of async/await and Promise.all allows for writing clean, readable code that handles parallel asynchronous operations.

The system contains good and robust error handling, especially important for network requests and database querying.

Non-Functionalities

Goal 1: **Effective Resource Usage**: this goal involves optimizing the usage of computational resources (such as CPU), ensuring that the application scales appropriately based on different demand of these resources, and avoiding overconsumption of resources, leading to inefficiency. We want the system to dynamically allocate resources based on real-time needs. For example, different images may require varying levels of processing power due to differences in resolution, number of heads, complexity of the image, … another example could be that the frontend-backend service might experience spikes in traffic during peak hours, requiring more resources to handle increase user request and maintaining performance. So the system should be able to automatically scale the resources, but on the contrary during off-peak times, the resource demand may decrease, and when you then maintain a large number of active instances the could become costly.

This is managed by Kubernetes feature of Horizontal Pod Autoscaling (HPA), which automatically adjust the number of active pods based on for example CPU usage or any other metric. During high demand this will increase the workload which will automatically scale the components pods by increasing the number of pod replicas in a deployment. I will also set limits for the resource requests for each pod, to not over demand resources, which could decrease the performance for the rest of the application. When the HPA increases the pods, the load balancer will handle distributing the requests to these pods.

I have set up 2 horizontal pod autoscaling,

- one on the frontend/backend component (when lots of user want to connect/request at the same time), this will be tested by using “autocannon -c 100 -d 60 http://127.0.0.1:15162/” which initiates 100 concurrent connections to the URL http://127.0.0.1:15162/ for a duration of 60 seconds, when we perform this we can see there are automatically new frontend/backend pods generated and is still efficiently accessible from the browser. This can be seen in demo 2.
- one on the image prediction software (when there are a lot of images that needs to be processed at the same time), this will be tested by spamming the upload buttons of different clients of different upload zones many times after each other, which can be seen in demo 3.

Goal 2: **Fault tolerance**: because the system could fail because of different reasons for example: corrupt image (and the image prediction component goes down), high traffic during peak (frontend/backend component could get overwhelmed), …. For this Kubernetes comes useful because of the distributed deployed application.

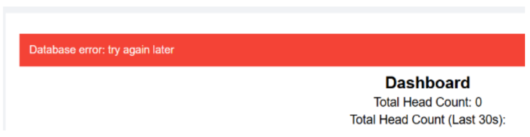
In demo 4, we can see for example when we let a pod go down manually the application still works. Kubernetes will make this pod restart and try to minimize down time, when we have multiple pods, and a pod goes down the load balancer also helps for fault tolerance by letting all requests go to the alive pod and not the crashed pod. Because of this deployment strategy with Kubernetes the failures in different components are isolated. Each service (frontend/backend, image prediction, database) runs in separate containers and pods. This isolation inherently helps in containing crashes. For instance, if the image prediction service crashes, it won't directly impact the frontend/backend or database services.

The setting in the deployment "resource limits and requests", define resource limits and requests for each container to prevent a single service from consuming all available resources, which can starve other services.

Use of good error handling and informing the user of errors/down components is also import for fault tolerances:

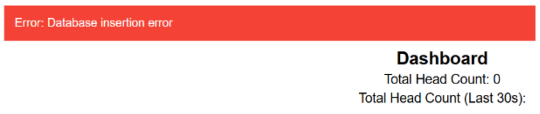
- When the frontend-backend component and image prediction component is alive, but MySQL component is not alive:

When a client connects:


Database error: try again later

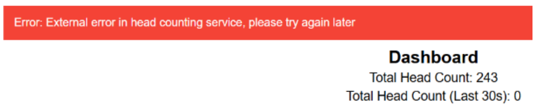
Dashboard
Total Head Count: 0
Total Head Count (Last 30s):

When a client is connected and wants to upload an image:


Error: Database insertion error

Dashboard
Total Head Count: 0
Total Head Count (Last 30s):

- When the front-backend component and mysql component is alive, but image prediction component is not alive:


Error: External error in head counting service, please try again later

Dashboard
Total Head Count: 243
Total Head Count (Last 30s): 0

when the client connects and tries to upload an image, it gets error (the mysql functionality works: the last uploaded image, headcount, last headcount are displayed)

Future Work

By addressing both effective resource usage and fault tolerance, the system is robust, scalable, and reliable, capable of handling varying loads and recovering gracefully from failures.

- Best practices for a database? Replicas, and keeping all data synced over the replicas
  - Master to master or master to slave architecture
- Better monitoring with for example Grafana
- Implement security management
- Integrate the deployment in a CI/CD pipeline
- **Split frontend and backend or even containerize different city cameras separately**

Commands for running the application

Commands to run the application (starting in home directory of the application)

1. make sure docker desktop is running
2. minikube start
3. cd mysql-kube
4. kubectl apply -f mysql-pv.yaml
5. kubectl apply -f mysql-pvc.yaml
6. kubectl apply -f mysql-configmap.yaml
7. kubectl apply -f mysql-deployment.yaml
8. kubectl apply -f mysql-service.yaml
9. kubectl exec -it <pod name of the just generated pod> -- mysql -uroot -p (ww root)
10. USE images;
11. CREATE TABLE images_table (
    id INT PRIMARY KEY AUTO_INCREMENT,
    image LONGTEXT NOT NULL,
    creation_date DATETIME NOT NULL,
    upload_zone INT NOT NULL,
    head_count INT NOT NULL
    );
12. cd ..
13. cd image-predict
14. @FOR /f "tokens=*" %i IN ('minikube -p minikube docker-env --shell cmd') DO @%i
15. docker build -t predict-image .
16. minikube image ls
17. kubectl apply -f image-predict-deployment.yaml
18. cd ..
19. cd frontend-backend
20. @FOR /f "tokens=*" %i IN ('minikube -p minikube docker-env --shell cmd') DO @%i
21. docker build -t front-backend-image .
22. minikube image ls
23. kubectl apply -f frontend-backend-deployment.yaml
24. cd ..
25. minikube addons enable metrics-server
26. kubectl autoscale deployment frontend-backend --max=10 --min=1 --cpu-percent=70
27. kubectl autoscale deployment image-predict --max=10 --min=1 --cpu-percent=30
28. check: kubectl get all
29. to start on localhost (create a tunnel): minikube service –all

autocannon used in demo: autocannon -c 100 -d 60 http://127.0.0.1:15162/