FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Project Report

Automatic Pet Detection With Edge Computing

by

Group 2
Vincent Roßknecht
Jonas Hülsmann
Marco Tenderra
Minh Kien Nguyen
Alexander Atanassov

Supervisor
Prof. Dr. Christian Baun

Submission Date
July 14th, 2023

Table of Contents

Overview

Introduction: The project *Automatic Pet Detection With Edge Computing* is part of the Cloud Computing SS23 module of Prof. Dr. Christian Baun at the Frankfurt University of Applied Sciences.

Objective: This project aims to develop an edge computing solution for the automatic detection of cats and dogs. General steps to achieve the project goal are listed in the **Project Plan** part of this Overview section.

Duration: 12.04.2023 - 05.07.2023

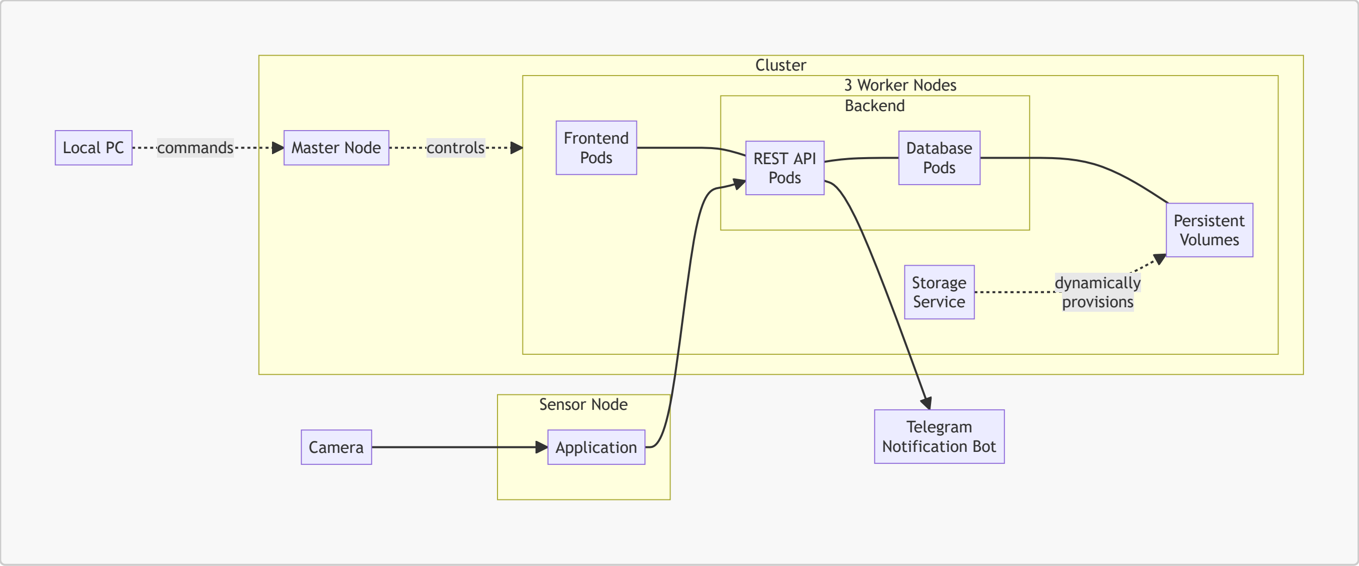
Source Code: Link

Presentation Slides: Link

Hardware:

Received from Prof.	Obtained from own source
1x Raspberry Pi 4 Model B (Pi 4B) 1x Raspberry Pi 3 Model B+ (Pi 3B+) 3x Raspberry Pi 3 Model B V1.2 (Pi 3B) 5x Samsung 32GB MicroSDHC 1x Apple USB-C-to-USB-C Charger 1x Anker 6-Port PowerPort 2x TP-Link TL-SG105 5-Port Desktop Switch 6x LAN Cable 4x CoolReal USB-C-to-USB-C Cable 1x Raspberry Pi Camera Module 2 (Camera Module)	1x FRITZ!Box 3272 Router 1x USB-to-USB-C Cable 1x ISY ICR-120 8-in-1 Card Reader

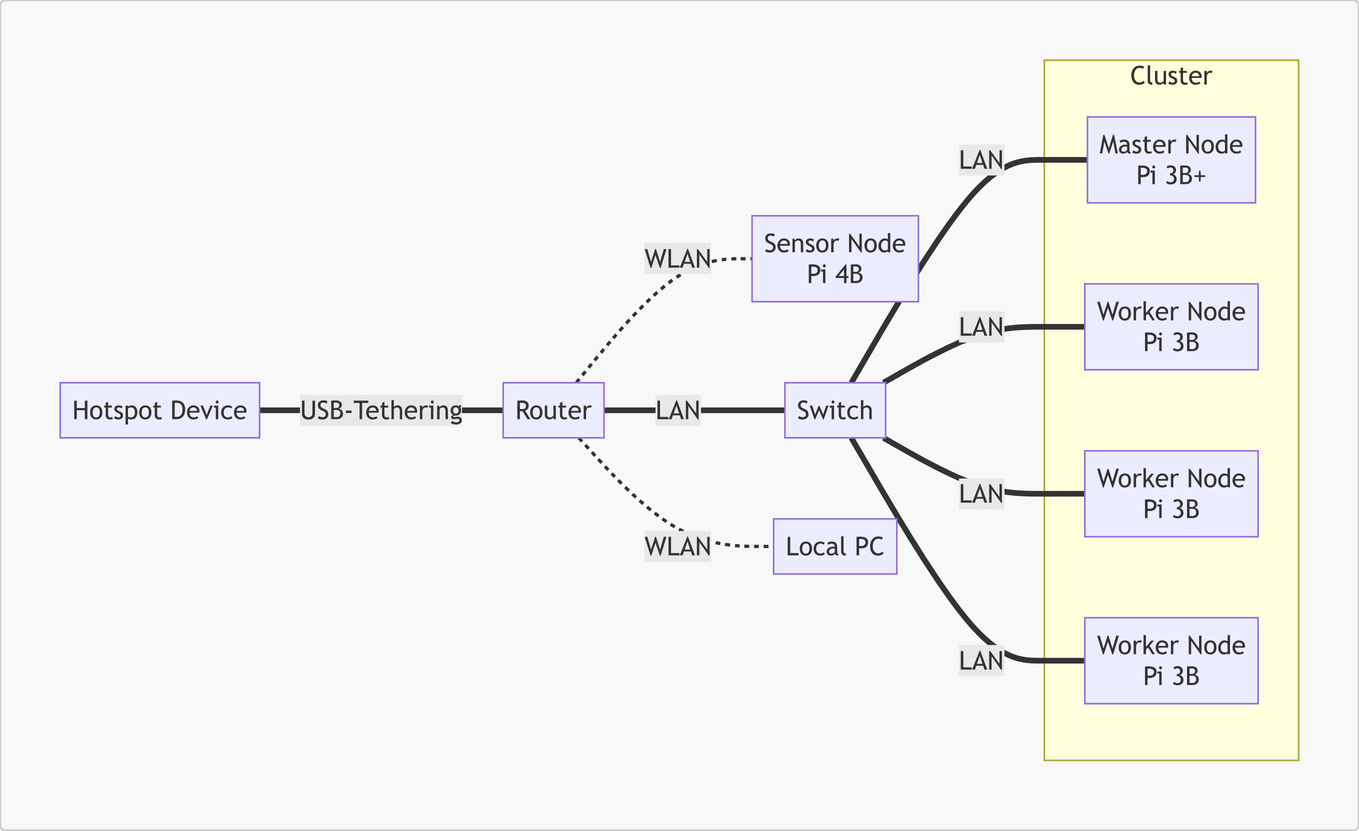
System Architecture:



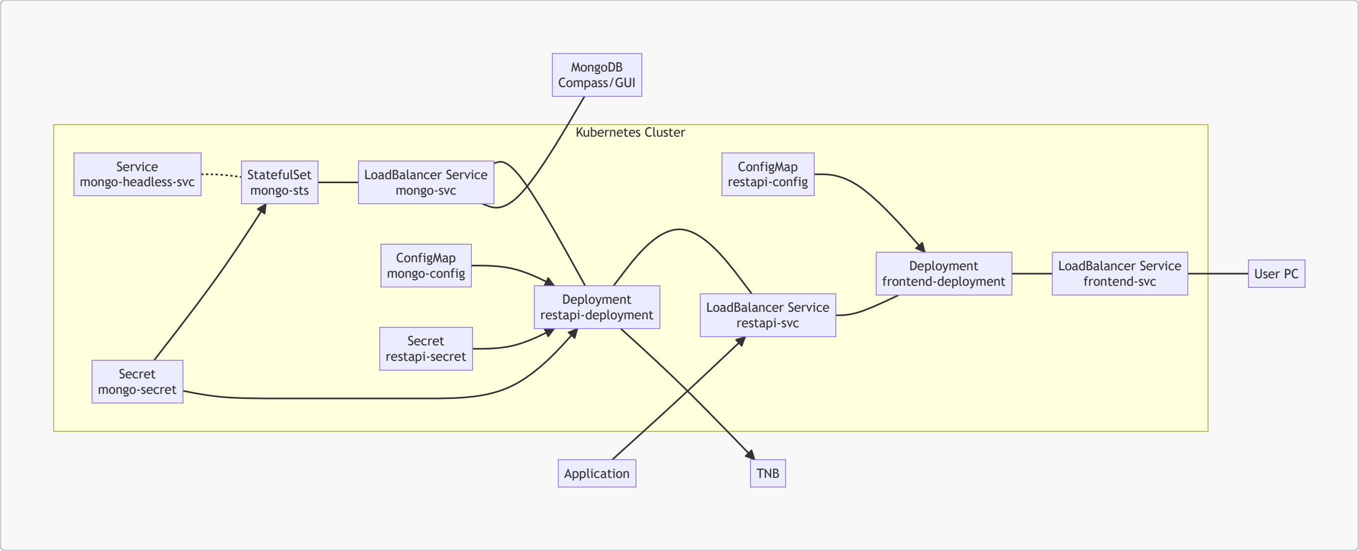
Component	Role
Camera	capture and send visual data to the sensor node
Application	- analyze visual data for pet detection - process and pack pet image & detection results into JSON format - send JSON data to the cluster
Persistent Volumes (PV)	- serve as persistent storage resource in the cluster - use local storage available on worker nodes
Storage Service	- dynamically provision PV - manage the underlying storage infrastructure of PV
Frontend Pods	- provide user interface - handle user interactions
REST API Pods	process data & facilitate data communication between system components
Database (DBS) Pods	- handle read- and write-requests (queries) for detection results - synchronize & replicate data across pods/worker nodes (Master-slave replication in DBS)
Telegram Notification Bot (TNB)	notify user about detection results via Telegram
Local PC	serve as tool for setting up system

System Behavior: See Test System section.

Network Architecture:

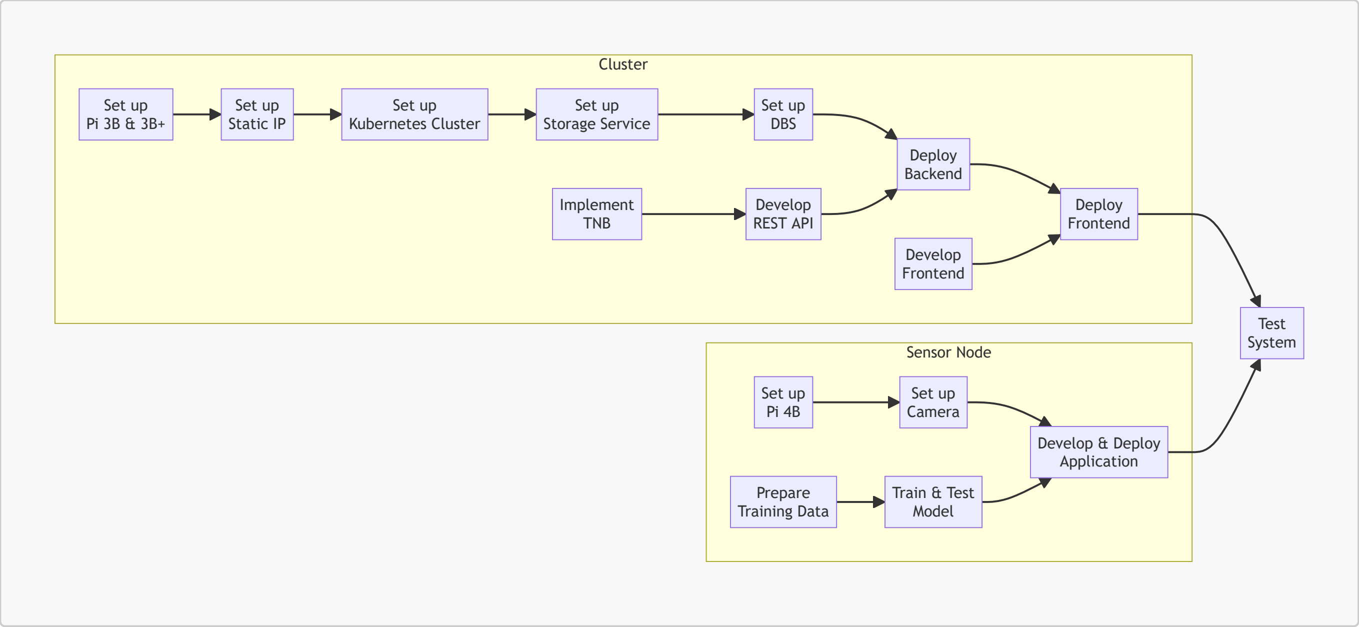


Kubernetes Architecture:



MongoDB Compass/GUI
Kubernetes Cluster
Service mongo-headless-svc
StatefulSet mongo-sts
LoadBalancer Service mongo-svc
ConfigMap restapi-config
ConfigMap mongo-config
Deployment restapi-deployment
LoadBalancer Service restapi-svc
Deployment frontend-deployment
LoadBalancer Service frontend-svc
User PC
Secret restapi-secret
Secret mongo-secret
Application
TNB

Project Plan:



Group 2 Info & Task Distribution:

Member MatrNr.	**Uni-Mail**	**Primary Tasks**	**Secondary Tasks**
Vincent Roßknecht 1471764	vincent.rossknecht@stud.fra-uas.de	- Prepare Training Data - Train & Test Model - Test System	
Jonas Hülsmann 1482889	jonas.huelsman@stud.fra-uas.de	- Develop REST API - Deploy Backend	- Test System
Marco Tenderra 1251463	tenderra@stud.fra-uas.de	- Set up Pi 4B - Set up Camera - Prepare Training Data - Develop & Deploy Application	- Develop REST API - Test System
Minh Kien Nguyen 1434361	minh.nguyen4@stud.fra-uas.de	- Set up Pi 3B & 3B+ - Set up Static IP - Set up Kubernetes Cluster - Set up Storage Service - Set up DBS - Implement TNB	- Deploy Backend - Deploy Frontend - Test System
Alexander Atanassov 1221846	alexander.atanassov@stud.fra-uas.de	- Develop Frontend - Deploy Frontend	- Develop REST API - Test System

Sensor Node

Set up Pi 4B

- Insert an empty SD-Card into local PC.
- Install then run Raspberry Pi Imager on local PC.
- In the Raspberry Pi Imager:
    - For Operating System, select `Raspberry Pi OS (64-bit)`.
    - For Storage, select the inserted SD-Card.
    - In Advanced options (Cog icon):
        - Set `pi0` as hostname.
        - Set `admin` as username and set own password.
        - Enable `Enable SSH` and `Use password authentication` options. This allows for remote access and control of Pi 4B via SSH from local PC.
        - Enable `Configure wireless LAN` option, type in the SSID and password of the router so that Pi 4B will automatically connect to the router network.
        - To save the above advance options for further use, set Image customization options to `to always use`.
    - Write to SD-Card.
- Connect and Start up Pi 4B with SD-Card.
- SSH into Pi 4B from local PC with the command `ssh admin@pi0.local`
- Update system packages with `sudo apt update` then `sudo apt upgrade -y`
- SSH only provides *terminal* access to Pi 4B. To *remotely control the desktop interface* of Pi 4B, we use VNC (Virtual Network Computing). To enable VNC connection:
    - First, enable VNC Server on Pi 4B. SSH into Pi 4B from local PC, then enter `sudo raspi-config`. Now with the arrows select `Interfacing Options`, navigate to `VNC`, choose `Yes`, and select `Ok`.
    - Install Real VNC Viewer on local PC.
    - Open local VNC Viewer, enter `pi0.local:0` or `[IP address of Pi 4B]`. To find the IP address of Pi 4B, SSH into Pi 4B from local PC, then enter `hostname -I`.
    - Enter login credentials that were set while configuring Raspberry Pi Imager.
    - The VNC session should start, and the Raspberry Pi desktop should be available.

Set up Camera

- To connect Camera Module to Pi 4B, follow the steps listed in Connect the Camera Module. Make sure the Camera Module faces the USB and Ethernet ports.
- To test if the connection is working, enter `libcamera-still -o test.jpg` to capture a single image. For more information about `libcamera-still`, refer to this documentation.

Prepare Training Data

- First, we downloaded unannotated cat and dog images from Kaggle.

- Next, we annotated these images with MegaDetector (*Note: To annotate an image means to add annotation files that contain the bounding boxes and types of the objects in the image*). The results is a JSON annotation file for all images. Since MegaDetector can only differentiate between `Animals`, `Humans`, and `Vehicles`, the downloaded cat and dog images are kept separated. Therefore we have two JSON files with the MegaDetector annotation: one for cats and one for dogs. For some images MegaDetector couldn't find an annotation, because the quality of the image wasn't good enough. In total the dataset has around 35.000 images, which should be sufficient for training.

```
dataset/
├── cats
|    ├── megaDetector.json
|    ├── cat_0.png
|    ├── cat_1.png
|    ├── ...
├── dogs
|    ├── megaDetector.json
|    ├── dog_0.png
|    ├── dog_1.png
|    └── ...
```

- Then, we converted the annotation format to the YOLOv8 format using the this script, after this the images are ready for training. The annotations are extracted from the two JSON files and are written into multiple TXT files. The YOLOv8 annotation format requires one TXT annotation file for every image. Furthermore, the annotation for the bounding box itself changes from MegaDetector (`<class> x_top_left_bbox, y_top_left_bbox, width_bbox, height_bbox`) to YOLOv8 (`<class> x_center_bbox, y_center_bbox, width_bbox, height_bbox`). More information on the YOLOv8 annotation can be found here.

```
dataset/
├── cats
|    ├── images
|    |    ├── cat_0.png
|    |    ├── cat_1.png
|    |    └── ...
|    └── annotation
|         ├── cat_0.txt
|         ├── cat_1.txt
|         └── ...
├── dogs
|    ├── images
|    |    ├── dog_0.png
|    |    ├── dog_1.png
|    |    └── ...
|    └── annotation
|         ├── dog_0.txt
|         ├── dog_1.txt
|         └── ...
```

- Finally, we splitted the dataset into training, validation and test images. The number of images and the split we used are:

Pet	Training	Validation	Test
Cat	13.875	1.816	1.740
Dog	14.782	1.871	1.848
Sum	28.657	3.687	3.588
Percentage	79.75%	10.27%	9.98%

Train & Test Model

Train Model

We chose the YOLOv8 model, since it is the best choice for object detection. A comparison between YOLOv8 and other models can be found here. The training and validation for the YOLOv8 model is done in Google Colab. First we need to setup the Google Colab notebook. To train a YOLOv8 model install `ultralytics`, this project was done with version 8.0.105.

```
!pip install ultralytics
import ultralytics
```

In addition, it is necessary to establish a connection with Google Drive to conveniently access the training and validation datasets.

```
from google.colab import drive
drive.mount('/content/drive')
```

When dealing with a large number of files in Google Colab, it is advisable to compress the datasets into ZIP files before uploading. It is also recommended to make three distinct ZIP files for the training, validation, and test datasets. After uploading them to Google Drive, the ZIP files can then be easily extracted using the `!unzip` command within the Google Colab notebook.

```
!unzip '/content/drive/pathToZipFile/train.zip'
!unzip '/content/drive/pathToZipFile/validate.zip'
!unzip '/content/drive/pathToZipFile/test.zip'
```

After this there should be 3 folders in the direct environment of the Google Colab Notebook. Now we can start training, for better performance choose a GPU runtime in Google Colab (Runtime -> Change runtime type). In this project we used a Nvidia V100 GPU as runtime type. We need to create a YAML file to provide the paths to the datasets. In this project it looks like that:

```
train: yolov8/data/train
val: yolov8/data/train
```

```
# number of classes
nc: 2

names: ['cat', 'dog']
```

To start training run the following command, all possible parameters are listed here.

```
!yolo task=detect mode=train model=yolov8s.pt data=path/to/dataset.yaml epochs=20
batch=-1 project=path/to/result_storage name=pets
```

We chose the `yolov8s` model as our base because it offers a balance between training speed and accuracy, which suits our needs effectively. Using a Nvidia V100 GPU the traing of the model took ~5min/epoch for a total of ~1h40min. The results from the training, including the model, can be found in the `project` directory, which is specified in the command before.

A comprehensive overview of training with YOLOv8 can be found here. The summary of our training results can be found here as images in `training_results.png` and `training_confusion_matrix.png` or as a table here in the `results.csv` file. Here is an explanation for the different metrics from the results:

- `train/box_loss` and `val/box_loss`: These metrics measure the discrepancy between predicted bounding box coordinates and the ground truth bounding box coordinates during training and validation, respectively.
- `train/cis_loss` and `val/cls_loss`: These metrics address class imbalance by quantifying the difference between predicted class probabilities and the true class labels during training and validation, respectively.
- `train/dfl_loss` and `val/dfl_loss`: These metrics handle the issue of long-tail distribution by evaluating the discrepancy between predicted class distributions and the ground truth class distributions during training and validation, respectively.
- `metrics/precision` and `metrics/recall(B)`: Precision measures the accuracy of positive predictions, while recall (sensitivity) calculates the ratio of correctly predicted positive samples to the total number of actual positive samples. Both metrics provide insights into model performance.
- `metrics/mAP50` and `metrics/mAP50-95(B)`: Mean Average Precision (mAP) at an IoU threshold of 0.50 and mAP across a range of IoU thresholds (from 0.50 to 0.95 with a step size of 0.05) measure the average precision of correctly localized and classified objects, providing comprehensive evaluations of model performance at different IoU thresholds.

The letter "B" in `metrics/recall(B)` and `metrics/mAP50-95(B)` specifies, that this is an object detection model, whereas "(M)" would specify a segmentation model.

Test Model

To estimate the model performance, there were some further tests done on it. For this we use the test dataset with images the model was neither trained or validated with. This dataset contains 3.589 more images of both cats (1.740) and dogs (1.848). The model was used to identify the pet on these images and return the pet and the bounding box for every image. With the python script `top1_mAP.py` here the Top-1-Accuracy (Top-1-Acc) and the mean average Precision (mAP) are calculated. For the mAP calculation we used the function `average_precision_score` from the python package `sklearn`. The results are Top-1-Acc = 87.68% and mAP = 96.983%.

Develop & Deploy Application

Description

On the Sensor Node itself is *the* Application to take an image, process it, and send it to the backend. Six classes have been created for that purpose:

- `Camera`
- `Detection`
- `Package`
- `Compress`
- `Network`
- `SensorNode`

The Application code is object-oriented and written in `Python3`.

Camera

The `Camera` class imports the package `picamera2`, which can run on `64-bit` systems. The class has three methods. The methods `take_image` and `take_array` capture the current visual data in front of the Camera and return them as a `PIL` image or a `numpy` array, respectively. The last method is `stop_camera`, which stops the recording of images and frees up resources.

Detection

The `Detection` class analyses images by using the package `ultralytics`, which provides methods for loading and applying a YOLO model. During the initialization of a `Detection` object, the path to the trained model is passed as a parameter to load that model into the object. The model needs only be loaded once, thereby avoiding resource wastage. The class uses the method `make_prediction` for pet detection. More specifically, in this method the trained model is called to analyse images and detect pet(s). The method returns either successful detection results or a custom `NoBoundingboxDetected` exception indicating that no pet(s) could be detected.

Package

The `Package` class packs the results of the `Detection` class into JSON format. At the same time, the original input image is also processed: every detected object (pet) on the image will be framed inside a green bounding box and assigned a unique (identifier) number. For that purpose, the `opencv2` and `numpy` packages are used. To execute these steps, only the method `toJson` needs to be called.

Compress

The `Compress` class is tasked with converting the pet image output by the `Package` class into a `base64` string and putting it into the JSON data. For that purpose, the class provides the methods `compress_jpg` and `compress_png`, which convert a ByteArray image into a JPG or a PNG, respectively. These methods require us to temporarily save the image to the hardrive and then load it again. We decided to quickly save the image to `dev/shm`, which is a folder in RAM. All data on RAM are volatile and will be discarded after restart. The class also provides a method named `toBase64` for converting an image into a `base64` string.

Sample JSON data after `Compress`ed:

```
{
 "picture": <Base64 encoded string of image>,
 "date": "2023-05-29",
 "time": "11:03:46",
 "detections": [
  {
   "type": "Dog", "accuracy": 0.9125242233276367, "bid": 1
  },
  ...
 ]
}
```

Network

The `Network` class is there to send JSON data (i.e., packages) to the cluster. It has a method `sendPost` which uses the package `request` to send a POST request to the designated URL on the cluster. The class also has one method called `_send` which first checks if there are any packages to send and how many. `_send` was implemented with a queue, which can be used to queue up requests and send the packages more packed to the cluster. This implementation could reduce some overhead and speed up the data transferring process.

Controller/SensorNode

The main file is called `sensorNode.py`. This file needs to be executed in order to use the Application. It has an unlimited loop, which uses `itertools.count()`, which automatically counts the iterations. The Application itself is a console application. It has `argparse` implemented to exchange important configuration at runtime without the need of recoding. We use the following `arguments`:

- `--model`, default is `model/best.pt`, Path to our Model
- `--url`, default is `http://192.168.178.201/mongo/input`, URL of our backend on the cluster
- `--conf`, default is `0.5`, it sets the lowest percentage of confidence our model should accept
- `--queue`, default is `1`, it sets the queue length in the `Network` class
- `--debug`, default is `false`, it saves the output image and output package
- `--single`, default is `false`, if true the application has only one iteration and then shuts down

For help, run the command `python sensorNode.py --help`

For a quick execution, run `python sensorNode.py`

Cluster

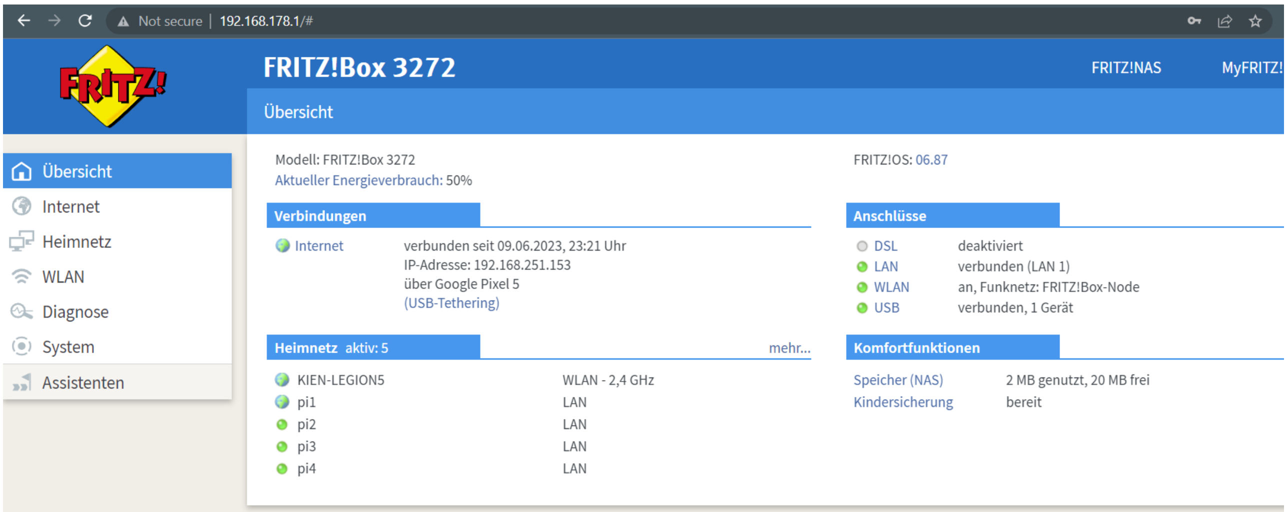
Set up Pi 3B & 3B+

- Follow the steps listed in Set up Pi 4B, but disable `Configure wireless LAN` option, and **DO NOT SSH into each Pi 3 yet!**
- For Operating System, select `Raspberry Pi OS Lite (64-bit)`.
- Set `pi1` as hostname for Pi 3B+, and `pi2`, `pi3`, `pi4` as hostname for each of three available Pi 3B.

Set up Static IP

For a Kubernetes cluster to work, the worker nodes must know the IP address of the master (controller) node and vice versa, so that they can communicate with each other. If the nodes' IP addresses change during communication, the Kubernetes cluster won't work. It is therefore critical that the master and worker nodes be assigned static (fixed) IP addresses. For that purpose, we use an additional FRITZ!Box Router. Here are the steps to set up static IP addresses:

- Turn on all hardware shown in the Network Architecture part of the Overview section.
- Share the hotspot device's internet connection with the router through USB-Tethering.
- Connect local PC and all Pi with the router network.
- On local PC, enter `ipconfig` on Command Prompt (in Windows) and look for the Default Gateway IP address of the router network (`192.168.178.1` in our case).
- Still on local PC, enter the IP address just found in a browser to open the router (FRITZ!Box) user interface (see below image; `KIEN-LEGION5` and `Google Pixel 5` were the local PC and hotspot device used, respectively).



Modell: FRITZ!Box 3272
Aktueller Energieverbrauch: 50%

FRITZ!OS: 06.87

Verbindungen

Internet verbunden seit 09.06.2023, 23:21 Uhr
IP-Adresse: 192.168.251.153
über Google Pixel 5
(USB-Tethering)

Anschlüsse

DSL	deaktiviert
LAN	verbunden (LAN 1)
WLAN	an, Funknetz: FRITZ!Box-Node
USB	verbunden, 1 Gerät

Heimnetz aktiv: 5 mehr...

KIEN-LEGION5	WLAN - 2,4 GHz
pi1	LAN
pi2	LAN
pi3	LAN
pi4	LAN

Komfortfunktionen

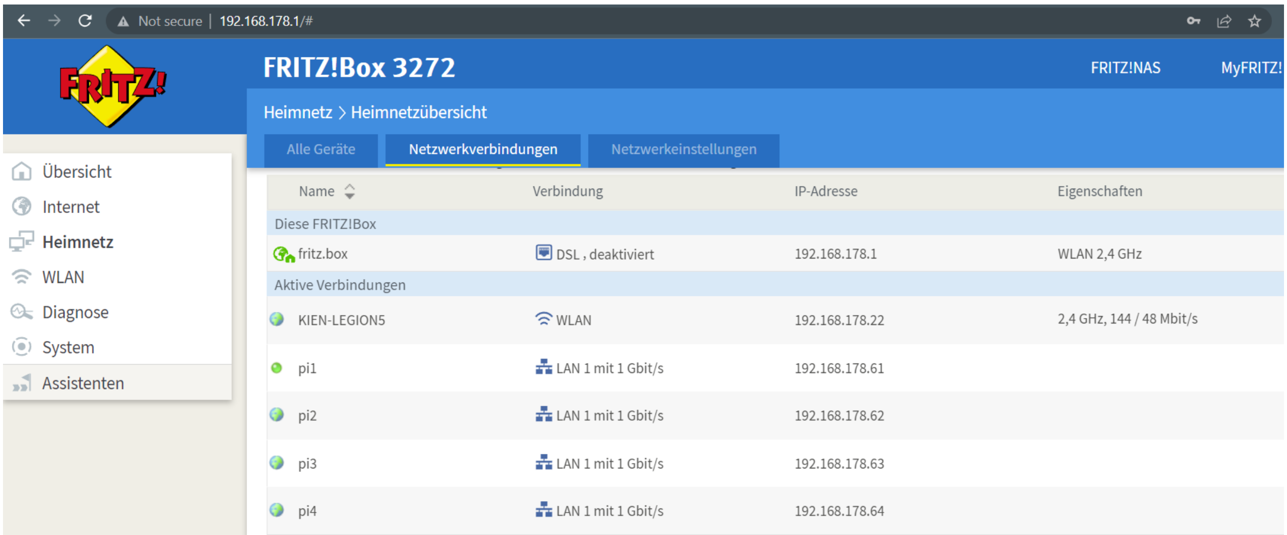
Speicher (NAS)	2 MB genutzt, 20 MB frei
Kindersicherung	bereit

- Assign static IP addresses to all available Pi, then restart all Pi.

Raspberry Pi	Assigned IP Address	Connection
`pi1`	`192.168.178.61`	LAN

Raspberry Pi	Assigned IP Address	Connection
`pi2`	`192.168.178.62`	LAN
`pi3`	`192.168.178.63`	LAN
`pi4`	`192.168.178.64`	LAN

- To check if the setup works, restart hotspot device, then share its internet connection again. All Pi should still have the same static IP addresses assigned to them.



Name	Verbindung	IP-Adresse	Eigenschaften
Diese FRITZ!Box			
fritz.box	DSL , deaktiviert	192.168.178.1	WLAN 2,4 GHz
Aktive Verbindungen			
KIEN-LEGION5	WLAN	192.168.178.22	2,4 GHz, 144 / 48 Mbit/s
pi1	LAN 1 mit 1 Gbit/s	192.168.178.61	
pi2	LAN 1 mit 1 Gbit/s	192.168.178.62	
pi3	LAN 1 mit 1 Gbit/s	192.168.178.63	
pi4	LAN 1 mit 1 Gbit/s	192.168.178.64	

After setting up static IP, for convenience we will enable passwordless, SSH-key-based login from local PC to each Pi 3:

- First, generate SSH key on local PC with:

```
# Do not fill anything when asked, just hit "Enter"
ssh-keygen -t rsa -b 2048
# Public key location: "~/.ssh/id_rsa.pub" (on Windows; "~" denotes home
directory)
```

- Then, copy the generated SSH key to each Pi 3 and finish setting them up. Do the following for each Pi 3:

```
# SSH into Pi 3, make sure you are user <admin>
# E.g., on local PC:
ssh admin@pi1.local

# Once logged in, go to <admin>'s home directory and create directory ".ssh"
cd
mkdir -p ~/.ssh

# Open new file "authorized_keys" in the ".ssh" directory
sudo nano ~/.ssh/authorized_keys
# Paste the contents of the public key "id_rsa.pub" into this file.
# Hit "Ctrl" + "X" -> "Y" -> "Enter" to save changes.

# Update system packages
sudo apt update && sudo apt upgrade -y

# Disable IPv6 & enable memory cgroup
sudo nano /boot/cmdline.txt
# Append "ipv6.disable=1 cgroup_memory=1 cgroup_enable=memory" at the end of the first line.
# It is important that there is no line break added.
# Hit "Ctrl" + "X" -> "Y" -> "Enter" to save changes.

# Reboot Pi 3 so all changes thus far take place.
sudo reboot
```

Set up Kubernetes Cluster

There are three possible designs for the Kubernetes cluster:

Design	Pros	Cons	Decision
1 Master & 3 Workers	- Simple setup - Enables fault tolerance & high availability in worker plane - Enables scalability across worker nodes	No fault tolerance & high availability in control plane	Adopt
2 Masters & 2 Workers	- Enables fault tolerance & high availability in both control & worker planes - Enables scalability across worker nodes	Complex setup	Discard
3 Masters & 1 Worker	Enables fault tolerance & high availability in control plane	- No fault tolerance & high availability in worker plane - Complex setup - No scalability across worker nodes	Discard

We prioritize *setup complexity* > *high availability & fault tolerance* > *scalability*, which is why we adopt the first design. Our Kubenetes cluster now consists of `pi1` as master node and `pi2, pi3, pi4` as worker nodes.

We first tried to set up the four given Pi 3 as a `K3s` cluster. However, huge CPU and MEM usage (100~300% and >65%, respectively) by `k3s-server` on fresh install made the master node barely respond to any command. The workarounds suggested in `K3s` documentation could not alleviate the problem for us. Hence, instead of `K3s`, we used `K0s`. Here are the steps to set up set up a `K0s` cluster:

- On `pi1` (the designated master node):
  - Run `curl -sSLf https://get.k0s.sh | sudo sh` to download the latest stable `K0s`.
  - Run the following commands to deploy as master (controller) node:

```
# Install, start, and check the k0scontroller service
sudo k0s install controller
sudo systemctl start k0scontroller.service
systemctl status k0scontroller.service
```

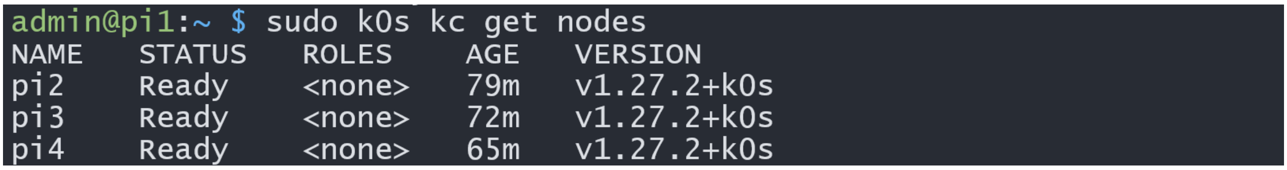
  - Create a token with which new worker nodes can join the `K0s` cluster by `pi1`. Save the join token for subsequent steps.

```
sudo k0s token create --role worker
```

- On each `pi2`, `pi3`, and `pi4` (the designated worker nodes):
  - Run `curl -sSLf https://get.k0s.sh | sudo sh` to download the latest stable `K0s`.
  - Run the following commands to deploy as worker node:

```
# To join the K0s cluster by pi1, create the join token file for the
worker
# $TOKEN_CONTENT is the join token created by pi1:
sudo sh -c 'mkdir -p /var/lib/k0s/ && umask 077 && echo
"$TOKEN_CONTENT" > /var/lib/k0s/join-token'

# Install, start, and check the k0sworker service
sudo k0s install worker --token-file /var/lib/k0s/join-token
sudo systemctl start k0sworker.service
systemctl status k0sworker.service
sudo k0s status
```

- Run `sudo k0s kc get nodes` on `pi1` to verify if the whole setup works. Note that `pi1` is not shown, because by default `K0s` only lists nodes with workloads, i.e., worker nodes.



```
admin@pi1:~ $ sudo k0s kc get nodes
NAME    STATUS   ROLES    AGE   VERSION
pi2     Ready    <none>   79m   v1.27.2+k0s
pi3     Ready    <none>   72m   v1.27.2+k0s
pi4     Ready    <none>   65m   v1.27.2+k0s
```

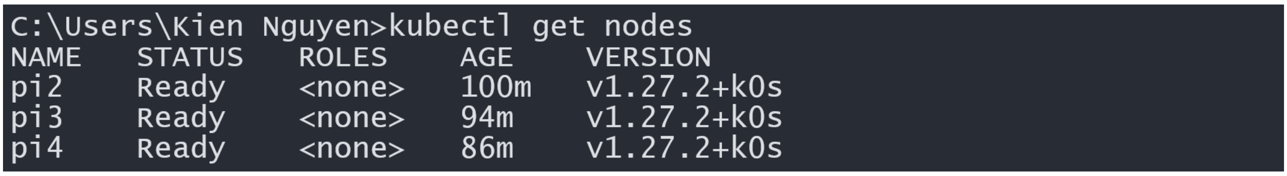
For convenience we will install `Helm` and configure `kubectl` on local PC. `Helm` is the package manager for Kubernetes, and `kubectl` is the Kubernetes command-line tool that allows us to run commands against Kubernetes clusters.

- This guide shows how to install `Helm` on local PC.
- This guide shows how to install `kubectl` on local PC.
- To configure `kubectl` on local PC, open the file `/var/lib/k0s/pki/admin.conf` on `pi1` with `sudo cat /var/lib/k0s/pki/admin.conf` and copy its content.

- Paste the copied content in the `config` file normally available at `~/.kube/config` (`~` denotes home directory on local PC; if `.kube/config` is unavailable, create one). Here it is crucial to replace `localhost` in `clusters:cluster:server` with the static IP address of the master node (`192.168.178.61`). Everything else can stay the same.

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: ...
    server: https://192.168.178.61:6443
...
```

- Now we can access the setup `K0s` cluster from local PC. For example:



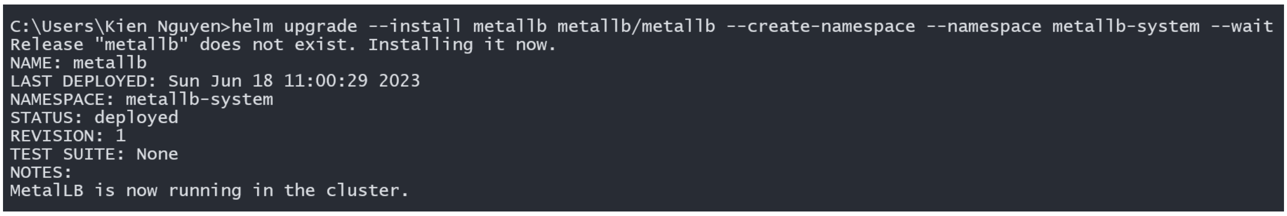
```
C:\Users\Kien Nguyen>kubectl get nodes
NAME    STATUS   ROLES    AGE    VERSION
pi2     Ready    <none>   100m   v1.27.2+k0s
pi3     Ready    <none>   94m    v1.27.2+k0s
pi4     Ready    <none>   86m    v1.27.2+k0s
```

As preparation for future tasks we will install and configure `MetalLB`, which exposes Kubernetes `LoadBalancer` services from our `K0s` cluster to applications/services outside of it.

- First, install `MetalLB`:

```
# Add metallb repository to helm
helm repo add metallb https://metallb.github.io/metallb

# Install metallb
helm upgrade --install metallb metallb/metallb --create-namespace --
namespace metallb-system --wait
```

Expected installation result:



```
C:\Users\Kien Nguyen>helm upgrade --install metallb metallb/metallb --create-namespace --namespace metallb-system --wait
Release "metallb" does not exist. Installing it now.
NAME: metallb
LAST DEPLOYED: Sun Jun 18 11:00:29 2023
NAMESPACE: metallb-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
MetalLB is now running in the cluster.
```

- Then, configure `MetalLB` by applying the `metallb.yaml`-script in the project source code. In the script we specify the IP address pool that `MetalLB` can assign to Kubernetes services of type `LoadBalancer` (from `192.168.178.200` to `192.168.178.220`), allowing these service to be accessible from outside the cluster.

```
# On local PC, change directory to script location, then
kubectl apply -f metallb.yaml
```

Expected configuration result:

```
ipaddresspool.metallb.io/default-pool created
l2advertisement.metallb.io/default created
```

Set up Storage Service

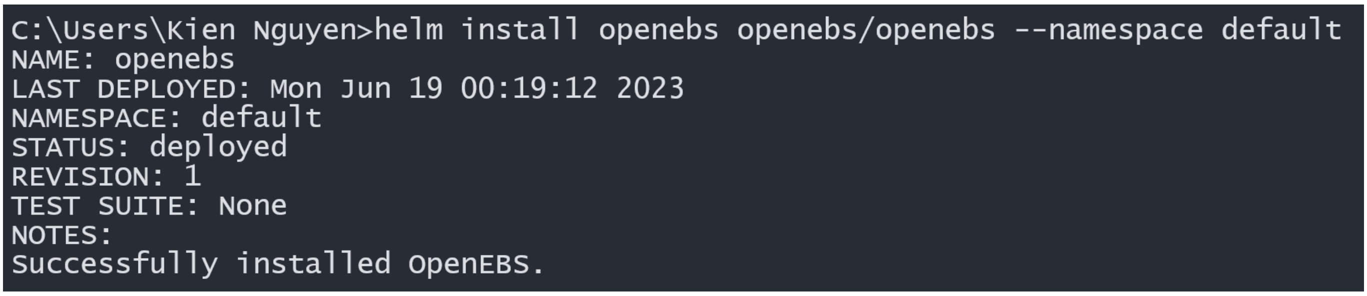
Initially, we wanted to use a storage service that can replicate data on PV across worker nodes, as this replication would provide high availability and fault tolerance for data on our `K0s` cluster. We tried using the lightweight `Longhorn` for that purpose (A comparison between `Longhorn` and several other storage services can be found here). However, after installation of `Longhorn`, our pods were repeatedly in `CrashLoopBackOff` status. Since we could not determine the exact error cause, and did not want to go over the complex prerequisites of `Longhorn` again for debugging, we abandoned `Longhorn` and tried the easier-to-set-up `OpenEBS` instead.

`OpenEBS` uses the storage available on Kubernetes worker nodes to provide Stateful(Set) workloads with Replicated Volumes, which is what we wanted initially. However, when we tried to use `OpenEBS Jiva Operator` (the only storage engine compatible with our hardware) for the provision of Replicated Volumes, our pods were also repeatedly in `CrashLoopBackOff` status. The same case happening with both `Longhorn` and `OpenEBS Jiva` made us conclude that using a storage service on our `K0s` cluster to replicate PV data is not recommendable. One possible reason is such data service would add overhead on the cluster capacity and performance, eventually leading to out-of-memory or -resource, which is one of the common causes for `CrashLoopBackOff`.

We therefore delegate the replication of PV data across worker nodes to the multiple DBS Pods running in our `K0s` cluster, as these pods (each running on a worker node) would have to synchronize their PV data to ensure data consistency anyway. We employ `OpenEBS` as a storage service that only serves to dynamically provision local PV for the DBS Pods. For that purpose, `OpenEBS` provides OpenEBS Dynamic Local PV Provisioner and OpenEBS Local PV Hostpath. So that these resources can be used later, install `OpenEBS` with `Helm` as follows:

```
# Get repo info
helm repo add openebs https://openebs.github.io/charts
helm repo update

# Install
helm install openebs openebs/openebs --namespace default
```

Expected installation result:



```
C:\Users\Kien Nguyen>helm install openebs openebs/openebs --namespace default
NAME: openebs
LAST DEPLOYED: Mon Jun 19 00:19:12 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Successfully installed OpenEBS.
```

Set up DBS

Pre-API-Development Setup

Since we delegate the replication of PV data to the DBS Pods, we must use a DBS that enables data replication across its instances. That DBS must also support `arm64/v8` architecture on our Pi 3. Another important factor to consider is which type of DBS (relational or NoSQL) to be used for storing images and detection results, as these data will be queried later by users. Hence for each DBS type to consider, we pick a representative DBS that satisfies the above necessary conditions, then compare their characteristics:

`MySQL` **(Relational DBS)**	`MongoDB` **(NoSQL Document DBS)**
Complex replication setup	***Simple*** replication setup
Image data stored as BLOB, requiring ***less*** storage space	Image data stored as base64-encoded string, requiring ***more*** storage space
Detection data stored in tables, producing ***possibly quicker*** query results	Detection data stored in JSON documents, producing ***possibly slower*** query results
More work needed in REST API Pods to produce write-queries	***Less*** work needed in REST API Pods to produce write-queries

Since we prioritize *setup complexity* `>` *performance*, `MongoDB` is our choice for DBS. Here are the steps to set up `MongoDB` in our `K0s` cluster:

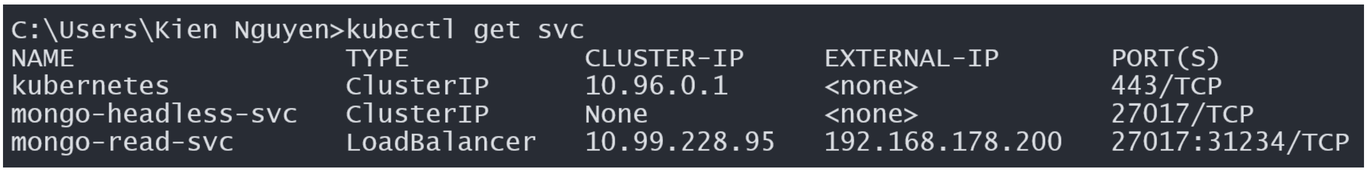
- Apply the following scripts in the project source code. After applying ensure that all corresponding pods are `Running` and all correspoding PV as well as Persistent Volume Claims (PVC) are `Bound`. For more information, read the scripts.

```
# On local PC, change to script directory, then apply scripts as follows
kubectl apply -f mongoSecret.yaml
kubectl apply -f mongoConfig.yaml
kubectl apply -f mongo.yaml

# Check if all corresponding pods are Running
kubectl get pods

# Check if all corresponding PV and PVC are Bound
kubectl get pv
kubectl get pvc
```

- Set up replication in MongoDB. The following commands are based on this tutorial:

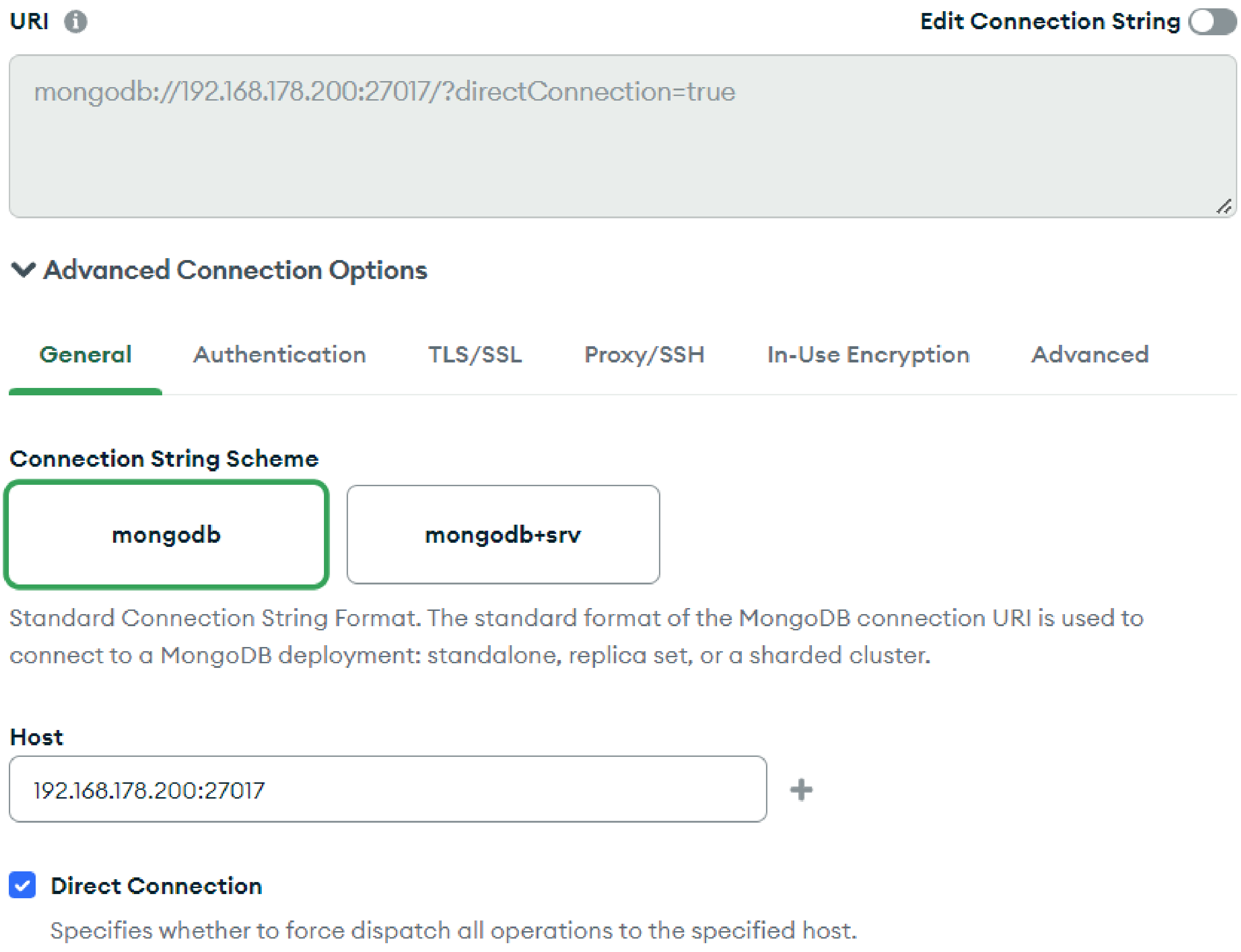
```
# On local PC, go into the first MongoDB server/pod "mongo-sts-0"
kubectl exec -it mongo-sts-0 -- mongo

# Initiate a replica set named "rs0" with the available MongoDB servers/pods
"mongo-sts-0", "mongo-sts-1", & "mongo-sts-2"
# "mongo-sts-0" will be set as the primary instance, while the other will be
set as secondary instances
rs.initiate(
   {
      _id: "rs0",
      version: 1,
      members: [
         { _id: 0, host : "mongo-sts-0.mongo-headless-
svc.default.svc.cluster.local:27017" },
         { _id: 1, host : "mongo-sts-1.mongo-headless-
svc.default.svc.cluster.local:27017" },
         { _id: 2, host : "mongo-sts-2.mongo-headless-
svc.default.svc.cluster.local:27017" }
      ]
   }
)

# Exit from "mongo-sts-0"
exit

# Go into "mongo-sts-0" again to check the initiated primary and secondary
instances
kubectl exec -it mongo-sts-0 -- mongo
rs.status()

# Enable replication from primary to secondary instances
rs.secondaryOk()
```

For convenience we will set up `MongoDB Compass/GUI`, so that we can check which data are available on our MongoDB database without having to go into a MongoDB server/pod. Since we use the `LoadBalancer` type for the Kubernetes Service `mongo-read-svc`, `MetalLB` will automatically assign a fixed IP address (`192.168.178.200` in our case) to this service, enabling `MongoDB Compass/GUI` to access `mongo-read-svc` and the MongoDB database from outside the cluster.



```
C:\Users\Kien Nguyen>kubectl get svc
NAME                 TYPE           CLUSTER-IP      EXTERNAL-IP       PORT(S)
kubernetes           ClusterIP      10.96.0.1       <none>            443/TCP
mongo-headless-svc   ClusterIP      None            <none>            27017/TCP
mongo-read-svc       LoadBalancer   10.99.228.95    192.168.178.200   27017:31234/TCP
```

In `MongoDB Compass/GUI`, configure the connection string as follows to enable connection:



mongodb://192.168.178.200:27017/?directConnection=true

Connection String Scheme

mongodb

mongodb+srv

Standard Connection String Format. The standard format of the MongoDB connection URI is used to connect to a MongoDB deployment: standalone, replica set, or a sharded cluster.

Host

192.168.178.200:27017

☑ Direct Connection

Specifies whether to force dispatch all operations to the specified host.

Mid-API-Development Setup

Initially, for the REST API Pods to write data to the Primary `MongoDB` instance (the only one in the replica set receiving write operations/requests), they would first have to send a read request to `mongo-read-svc` to query for the DNS name of that instance (e.g., `mongo-sts-0`). Only then can the REST API Pods send their write requests to the Primary `MongoDB` instance's DNS address (e.g., `mongo-sts-0.mongo-headless-svc.default.svc.cluster.local:27017`), which is exposed to them by the Kubernetes Service `mongo-headless-svc`. However, during the development of REST API, we were unable to get the DNS name of the Primary `MongoDB` instance while querying for it. Since the debugging process could not produce any significant results and we did not have enough time to consider other DBS options, we had to discard the `MongoDB Replica Set` setup on the cluster (i.e., the *Pre-API-Development Setup*) and went with only one `MongoDB` instance instead (i.e., the *Mid-API-Development Setup*), which enabled the system to function properly. As a consequence of changing the initial DBS setup, Test High Availability DBS will have to be skipped. However, from our perspective, the system's functionality takes precedence over the high availability of the DBS.

Here are the changes in the setup:

- There is now only one `MongoDB` instance (`mongo-sts-0`) on the cluster as opposed to three in the Pre-API-Development Setup.
- `mongo-read-svc` (assigned external IP: `192.168.178.200`), which was created initially to receive only read requests, was replaced with `mongo-svc` (assigned external IP: `192.168.178.204`), which currently receives both read and write requests, since there exists only one `MongoDB` instance to read from and

write to. This change is optional, as `mongo-read-svc` can also be configured to handle both read and write requests, but in that case the name of `mongo-read-svc` would not reflect exactly the types of request it receives.

Implement TNB

- Follow this tutorial to create a Telegram Bot. Our TNB is called `G2PetBot`.
- Create a Telegram group chat. Our group chat is called `Cloud Computing SS23`.
- Add `G2PetBot` to `Cloud Computing SS23`.
- Find the token of `G2PetBot` and the group ID of `Cloud Computing SS23`.
- Encode the token and group ID as `base64` strings.
- Add the encoded strings as values of `telegram-bot-token` and `telegram-group-chat-id` keys in the `restapiSecret.yaml`-script.
- Write code that sends detection results to TNB (see `telegram_bot/main.py` in the project source code). When `G2Petbot` receives detection results, it notifies all users in `Cloud Computing SS23` about them.

Develop REST API

Overview

The main tasks of the backend are to process data and to facilitate data communication between system components. More specifically, it provides an interface between the Sensor Node and the Frontend to send and retrieve data. The data sent and retrieved include the a `base64`-encoded picture, date and time of the picture, as well as detection results such as pet type and accuracy.

The programming language used for the implementation of the backend is `Python` in the version `3.10`. We have chosen `Python` over alternatives like `C++` or `Java`, because it is the only language every group member is equally familiar with. This means that everyone can help with his knowledge should problems during the development phase arise.

For the communication interface we have chosen `REST` as our framework over other alternatives like `MQTT` because of its ease of use and scalability as well as previously good experience with it.

Setup

- Download and install `Python 3.10` and an IDE of choice, e.g. `PyCharm`.
- Download and install necessary dependencies/libraries for the project to your `Python HOME` directory or virtual environment.
  - The dependencies are listed in the `requirements.txt` file in the `backend` and `flask` folders of the project source code.
  - Run `pip install -r requirements.txt` to install the dependencies

Django-REST-API

The first version of our backend has been implemented with the use of the `Django`-Framework. This framework provides us with a lot of built-in functionalities for the implementation of our REST-interface and the connection to our database.

The `Django` project folder consists of the following files:

- pics/
- Dockerfile
- db.sqlite3
- docker-compose.yml
- manage.py
- requirements.txt

manage.py is the entry-point to our backend and must be executed with the command python manage.py runserver from the command line to boot the web application.

Sqlite3 (db.sqlite3) is a database provided by the Django-framework. Until our final database solution was implemented, Sqlite3 served as a storage for the data utilized by the frontend for testing purposes.

requirements.txt, Dockerfile, and docker-compose.yml are all necessary config-files for the creation of a docker-image of the backend. The requirements.txt describes the required dependencies, the Dockerfile contains commands on how the image is supposed to be created, and docker-compose.yaml provides additional configuration.

The pics/ folder contains our internal program logic:

- settings.py contains our overall project configuration including application definitions, database access etc.
- models.py defines the database objects
- migrations/ folder contains all created instances of the models
- serializiers.py defines serializers based on the database object definitions of models.py. They are used to check if the correctness of the data sent by the camera module to backend.
- telebot.py implements a basic telegram-bot that posts a notification if a new picture has been sent to the backend by the camera
- urls.py defines the URLs used by the camera and the frontend to access the backend
- views.py implements the functionality of the urls.
    - Retrieve JSON data from the payload of the request
    - Check validity of the data via the serializers
    - save/retrieve data to/from the database
    - return a response to the client (including the retrieved data)

Once we decided to use MongoDB as our DBS, the Sqlite3 DBS was not needed anymore and a lot of functionalities became obsolete, because MongoDB is an object-storage DBS, whereas the serializers are designed for relational databases. Additionally, there were problems with the deployment of the Django-backend on the Kubernetes cluster. That is why we decided to drop Django and used the Flask-Framework instead.

Flask-REST-API

The new Flask-backend is more lightweight and comprehensible than the bloated Django-backend. The Flask project folder includes the following files:

- Dockerfile & requirements.txt for the docker image
- app.py as the entry point to the application and the definition of the REST-API-functionality
- mongo.py for creating a connection to the MongoDB on the Kubernetes cluster
- telebot.py for the telegram notification on camera input

This version of the backend has been successfully deployed on the Kubernetes cluster and provides a REST-interface for other applications inside and outside of the cluster.

Deploy Backend

The `Flask`-backend has been deployed by following these steps:

1. Create a docker image of the application with this command:

   ```
   docker build --platform linux/arm64 -t skywalker360/flask_pd:<tag>
   ```

   The `<tag>` is to be filled with the newest tag available + 0.1

2. Push the docker image to the Docker Hub repository

3. Apply the following scripts in the project source code:

   ```
   # On local PC, change to script directory, then
   kubectl apply -f restapiSecret.yaml
   kubectl apply -f restapiConfig.yaml
   kubectl apply -f restapi.yaml
   ```

Develop Frontend

Overview

The frontend is a web-application whose main task is to retrieve data from the backend and present them to the user. Data retrieved by the frontend are the captured images and their detection results. The frontend is also capable of retrieving data based on certain filter criteria. `Angular` was used as framework for the frontend.

Why Angular?

`Angular` is a TypeScript framework for interactive web-applications, meaning it provides a structure for developing user interfaces. There are two versions of `Angular`: `Angular` and `AngularJS`. The latter is older and for `JavaScript`, while the former is newer and for `TypeScript`. In this project `Angular` for `TypeScript` is used. Here are some advantages of `Angular` (for more information refer to this article):

- *Component-Based Architecture*: The application is splitted into smaller components which work together and can exchange information with each other. The components build a hierarchical structure. They are reusable and make the program more readable.

- *Two-way Data binding*: This helps the user to exchange data between the model (Typescript file) and the view (HTML file). This ensures that the model and view are alwayes sync.

- *Dependency Injection*: Dependencies are services or objects which are required for a class to work. Instead to create this objects inside the class, the class can request them. This reduces the coupling between components and services which is better for testability and maintainability.

- *Powerful Router*: `Angular` has a powerful navigation service which can load various components into the view depending on the URL in the browser.

Requirements

The web-application must be able to:

- Request captured images and detection results (date, time, pet id, type, accuracy) from the backend (Maximum 10 images per request) based on certain filter criteria
- Check for new captured images and detection results
- Navigate between the menu pages (In our case *Posts* and *About Us*)
- Display captured images and detection results

Setup

- Install `NodeJs` and `Angular CLI` (Windows)

    - Download and install `NodeJS` (JavaScript runtime environment).

    - Install `Angular` by running the following command in CMD:

      ```
      npm install -g @angular/cli
      ```

- Set up the project
    - Create a new project, e.g.:

      ```
      ng new web-app
      ```

    - Generate a component, e.g.:

      ```
      ng g component navbar
      ```

    - Generate new service:

      ```
      ng g service capture-loader
      ```

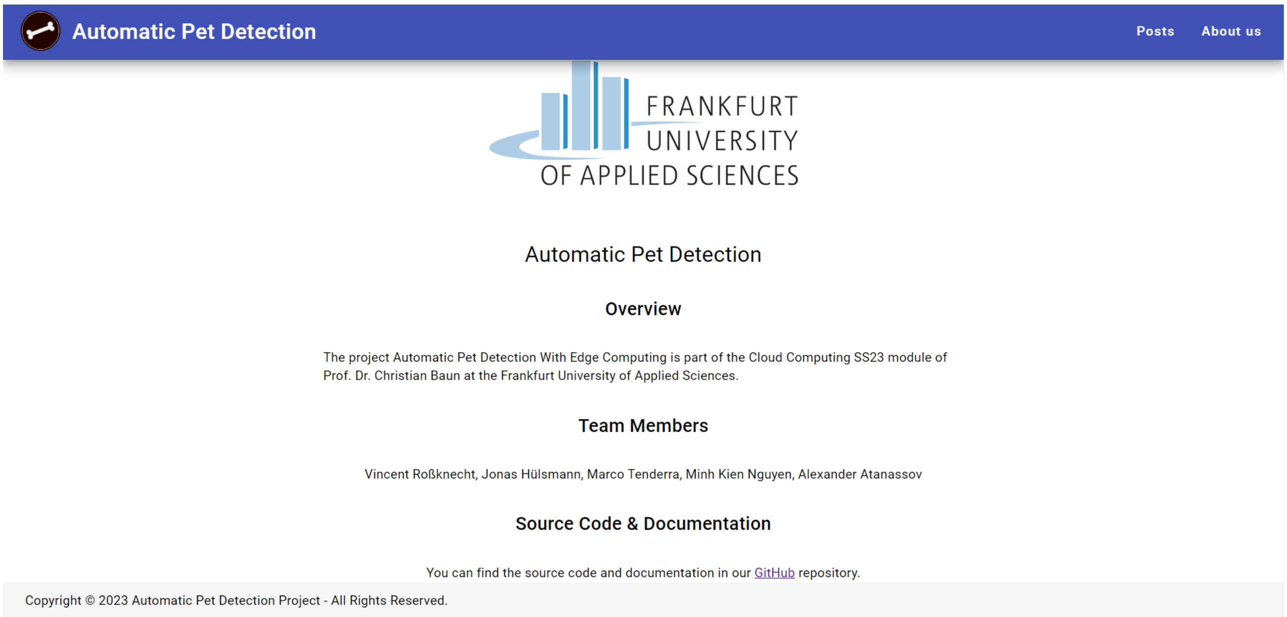
    - Run app:

      ```
      ng serve
      ```

      By default the app is hosted on localhost:4020.

Components

- *Navigation bar* enables users to navigate through the *Posts* and *About Us* pages of our app.
- *About Us page* displays general information about the project.



Automatic Pet Detection

Overview

The project Automatic Pet Detection With Edge Computing is part of the Cloud Computing SS23 module of Prof. Dr. Christian Baun at the Frankfurt University of Applied Sciences.

Team Members

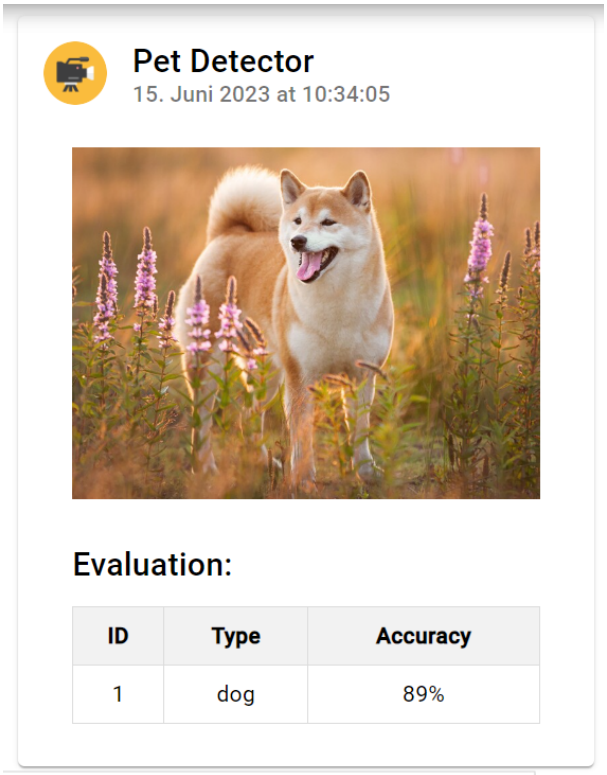
Vincent Roßknecht, Jonas Hülsmann, Marco Tenderra, Minh Kien Nguyen, Alexander Atanassov

Source Code & Documentation

You can find the source code and documentation in our GitHub repository.

Copyright © 2023 Automatic Pet Detection Project - All Rights Reserved.

- *Capture/Post* displays a single captured image and its detection results. The date and time of detection are listed first; below them are the captured image and a table which shows the id, type, and accuracy of every pet detection on the image. The component is used by the main page.



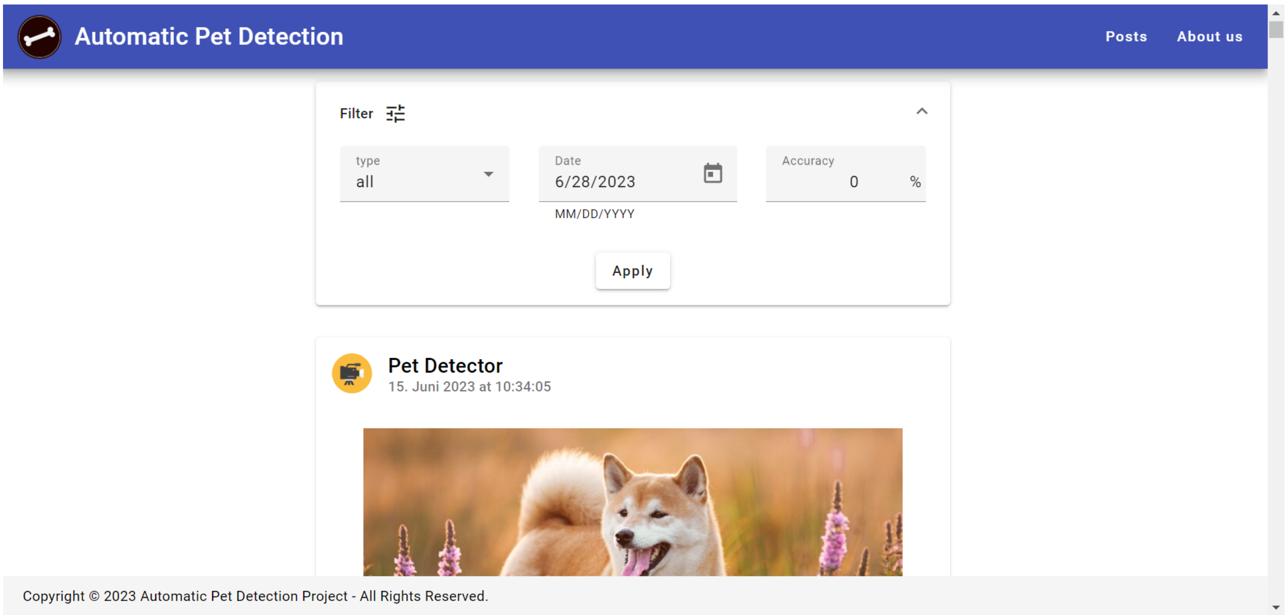
Pet Detector

15. Juni 2023 at 10:34:05

Evaluation:

ID	Type	Accuracy
1	dog	89%

- *Main page (Posts)* represents a scrollable list of Captures. By default, when the page is selected, 10 Captures (posts) are displayed. On a button click, 10 more are loaded. There is also a filter with which the user can specify the wanted pet type, earliest detection date time, and minimum accuracy.



type: all

Date: 6/28/2023 (MM/DD/YYYY)

Accuracy: 0 %

Pet Detector
15. Juni 2023 at 10:34:05

Copyright © 2023 Automatic Pet Detection Project - All Rights Reserved.

Service

A service is injected into the main page to retrieve data from the backend. For that purpose, the service makes the following HTTP requests:

- *Load Images (LI)*: LI request is used to load 10 captured images (and their respective detection results) from the backend. A filter is provided which specifies what criteria these images should match. The filter options are date (images must be before the given date), type (images must contain at least one pet of the given type), and accuracy (all pets on the images should have accuracy greater than or equal to the specified accuracy). LI request is also used to load the next 10 images from the backend. In this case, the ID of the last loaded image is also passed to the request so the backend can load the next images with the given filter.
- *Check New Images (CNI)*: This request checks if there are any new images in the database. The ID of the newest image on the frontend has to be provided so the backend can check if there are any new images in the database.

Deploy Frontend

Follow these steps to deploy frontend on the Kubernetes cluster:

- First, generate a Docker file so a docker image of the app can be created.
- Build project with production configuration:

```
ng build --configuration=production
```

- Generate Docker image for `linux/arm64` (To deploy the web-app on the Kubernetes cluster, a docker image for `linux/arm64` architecture must be created):

```
docker buildx build --platform linux/arm64 -t alllexander1/pets-app-arm64:v1
--push
```

- Apply the script `frontend.yaml`, which contains deployment configuration:

  ```
  # On local PC, change to script directory, then
  kubectl apply -f frontend.yaml
  ```

Test System

To verify that our system satisfies the project requirements and functions correctly (from the end user's perspective), we created the following test cases:

- Test TNB
- Test Main Functionality
- Test High Availability DBS

We designed each test case with *the IPO (Input-Process-Output) model* in mind. The IPO model provides a structured approach for identifying and defining the inputs, processes, and expected outputs of a particular functionality or process that needs to be tested.

Test TNB

Input: The user holds a dog / a cat / a dog or cat image in front of the Camera.

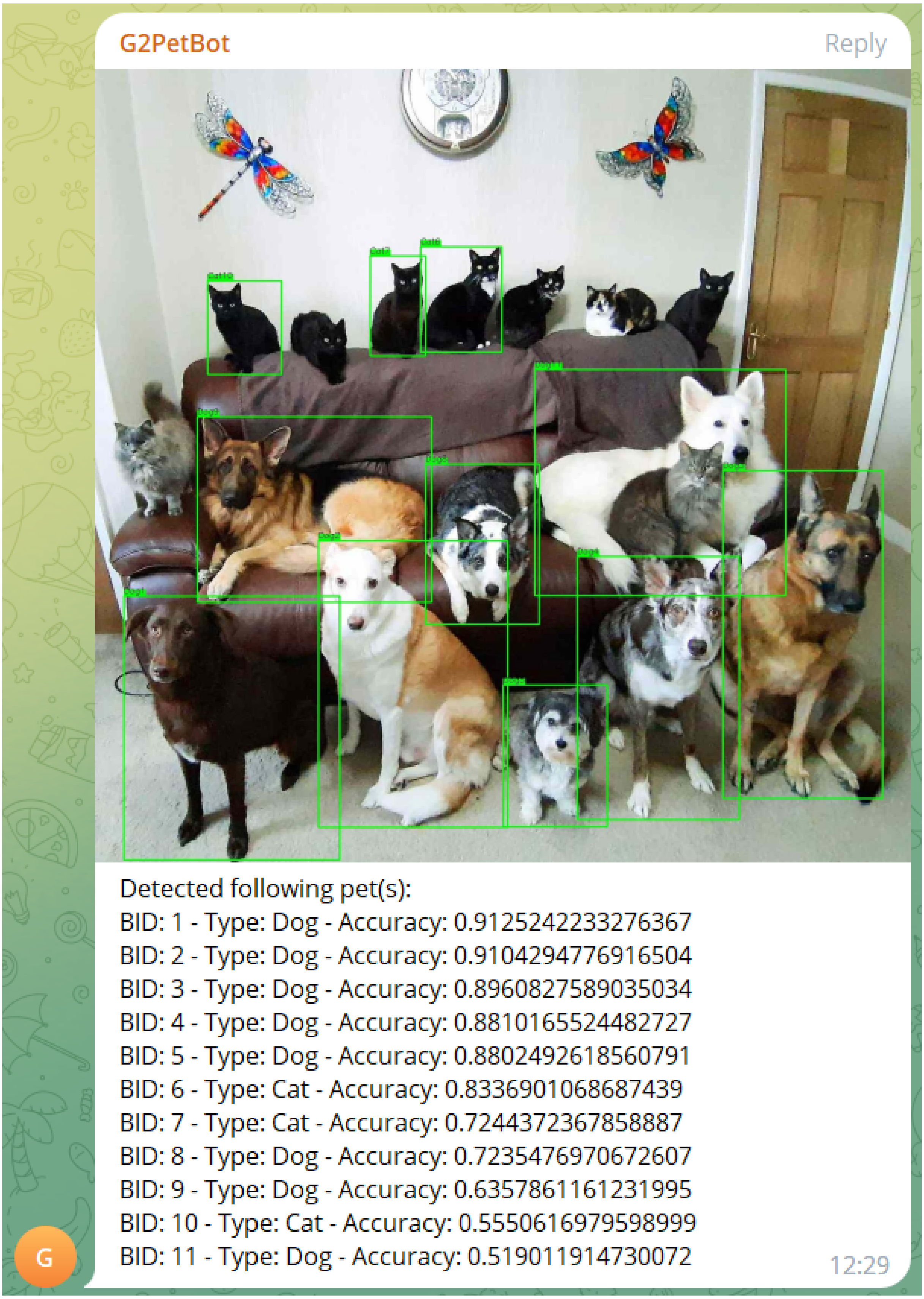
Process:

- The Camera continuously captures the visual data before it into images, then sends them to the Application in the Sensor Node.
- In the Application:
  - The `Detection` (Model) carries out pet detection constantly on the continuous stream of input images.
  - Upon successful detection, the resulting image and the corresponding detection results are forwarded to the `Package`, where the image is further processed and the detection results are packed into JSON format.
  - The *packaged* data are sent to the `Compress`, which encodes the processed image as `base64` string and puts it into the JSON results before sending them to the `Network`.
  - The `Network` forwards the *compressed* data to the Kubernetes Service `restapi-svc` on the cluster.
  - For more information about this process by the Application, see Develop & Deploy Application.
- Next, `restapi-svc` directs these data towards one of the REST API Pods running on one of the worker nodes. The REST API Pod receiving the data creates a notification from them and sends it to the TNB.
- Lastly, the TNB notifies the user about the pet image and detection results on Telegram.

Expected Output: The user receives a Telegram notification about the new pet image and detection results.

Current State: **PASSED**

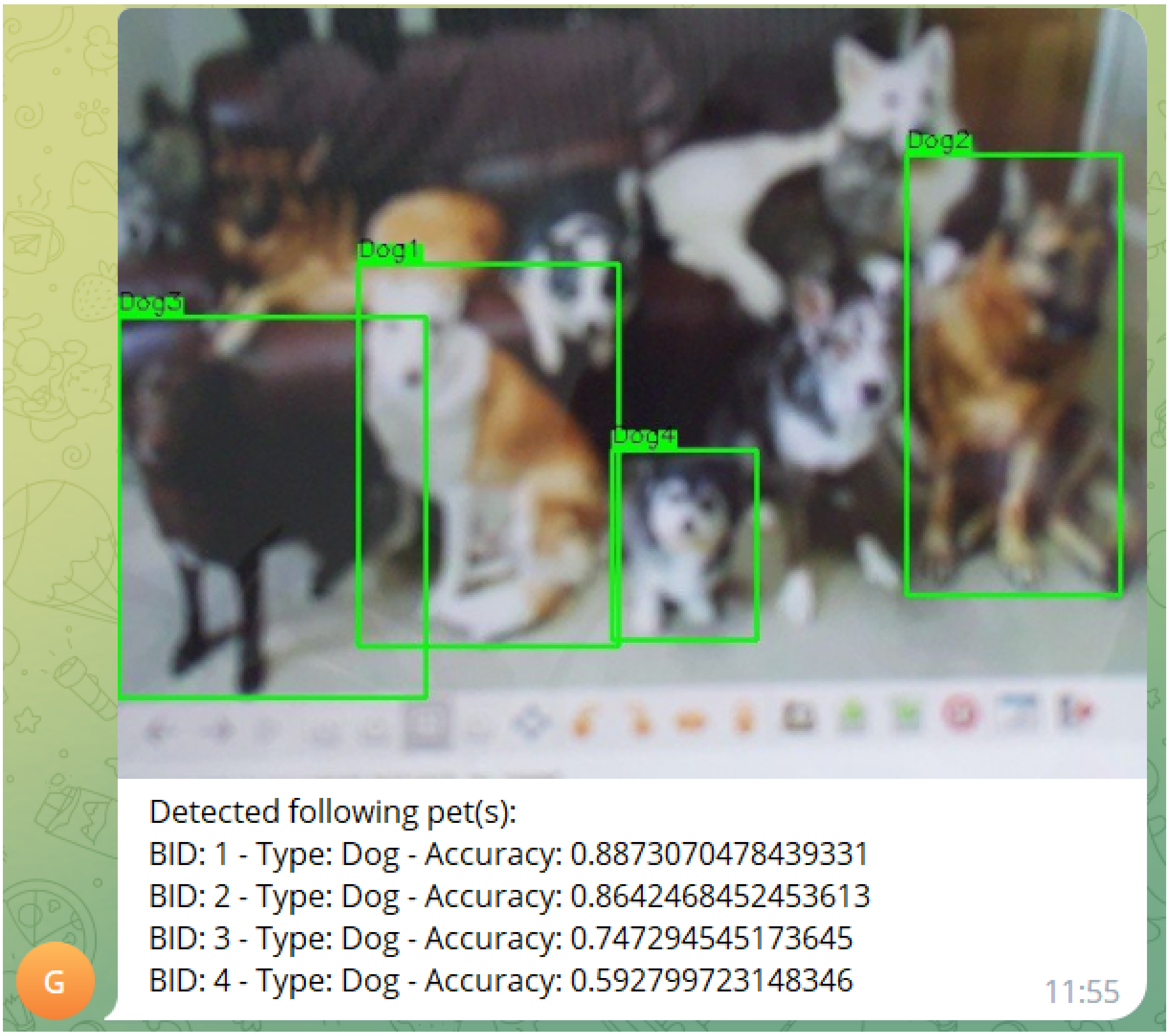
Actual Output:



G2PetBot

Detected following pet(s):
BID: 1 - Type: Dog - Accuracy: 0.9125242233276367
BID: 2 - Type: Dog - Accuracy: 0.9104294776916504
BID: 3 - Type: Dog - Accuracy: 0.8960827589035034
BID: 4 - Type: Dog - Accuracy: 0.8810165524482727
BID: 5 - Type: Dog - Accuracy: 0.8802492618560791
BID: 6 - Type: Cat - Accuracy: 0.8336901068687439
BID: 7 - Type: Cat - Accuracy: 0.7244372367858887
BID: 8 - Type: Dog - Accuracy: 0.7235476970672607
BID: 9 - Type: Dog - Accuracy: 0.6357861161231995
BID: 10 - Type: Cat - Accuracy: 0.5550616979598999
BID: 11 - Type: Dog - Accuracy: 0.519011914730072

12:29



Detected following pet(s):
BID: 1 - Type: Dog - Accuracy: 0.8873070478439331
BID: 2 - Type: Dog - Accuracy: 0.8642468452453613
BID: 3 - Type: Dog - Accuracy: 0.747294545173645
BID: 4 - Type: Dog - Accuracy: 0.592799723148346

Test Main Functionality

Input: The user interacts with the frontend UI to request certain data from the system.

Process:

- The Frontend Pod that provides the user with the frontend UI makes a HTTP request from the user's request (see **Service** part of the Develop Frontend section for more information). The HTTP request is then sent to the Kubernetes Service `restapi-svc`.
- Next, `restapi-svc` forwards that HTTP request to one of the REST API Pods, which then translates the HTTP request into a `MongoDB` query and sends the query to the Kubernetes Service `mongo-svc`.
- `mongo-svc` forwards the received query to the only `MongoDB` instance on the cluster, which handles the query by retrieving the requested data from its associated Persistent Volume.
- Finally, the requested data are passed back along the chain of communication to the Frontend Pod that received the user's request, which then displays the requested data on the frontend UI.

Expected Output: The data requested by the user are displayed on the frontend UI.

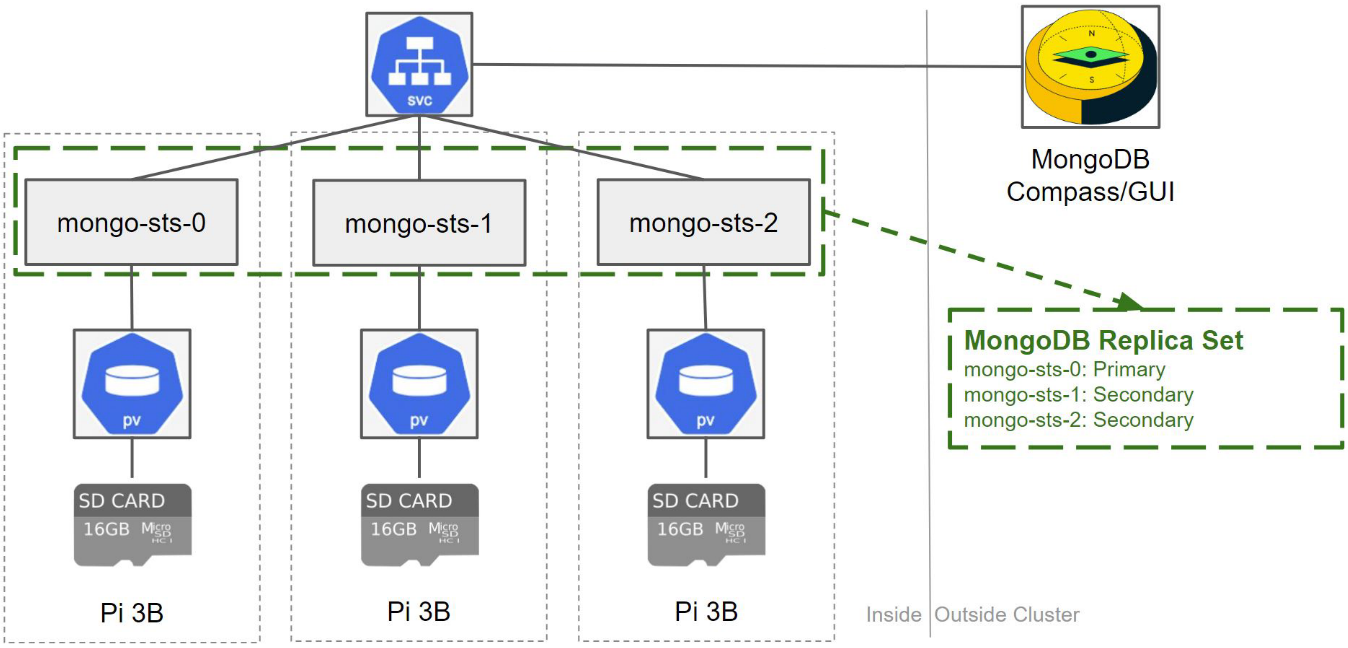
Current State: **NOT RUN** (will be updated)

Test High Availability DBS

Input: Failure of one of the worker nodes in the cluster, e.g. by unplugging it.

Process:

Each of the three worker nodes in our cluster hosts a StatefulSet instance of `MongoDB`. All three instances belong to the same `MongoDB Replica Set`. One of them is the Primary instance capable of performing both read and write operations (i.e., processing read and write requests); the other two are Secondary instances which can only carry out read operations. Thus, by default write requests are only sent to the Primary instance for processing.



When a worker node that hosts a Secondary instance fails:

- That instance no longer receives any read requests.
- Read requests are directed towards the Primary and the other Secondary instance.
- Write requests are still only sent to the Primary instance.

When the worker node that hosts the Primary instance fails:

- That instance no longer receives any read or write requests.
- One of the two Secondary instances will be elected as the new Primary instance.
- Read requests are sent to the new Primary and the only remaining Secondary instance.
- Write requests are exclusively directed towards the new Primary instance.

Expected Output: There is no changes in the system functionality from the user's perspective.

Current State: **SKIPPED** (see **Mid-API-Development Setup** part of the Set up DBS section for more information)