Rat Detection

FRANKFURT UNIVERSITY OF APPLIED SCIENCES
CLOUD COMPUTING - WS 2022/23
UNDER THE GUIDANCE OF PROF. DR. CHRISTIAN BAUN

EDITED BY

HAUCK, FABIAN (1367072)

KAISER, MARC (1434358)

KLIEMT, FREDERIK (1465987)

KNISS, HENRY(1226278)

MERKERT, LUCAS (1326709)

UNKART, CHRISTOPHER (1249284)

FEBRUARY 2023

Contents

1 Introduction

This project was completed in the cloud computing module of Prof. Dr. Christian Baun at Frankfurt University of Applied Sciences. Further references can be found on the course website [1]. This project aimed to develop an edge computing solution for the automatic detection of pests. Hence the rat detection represents an intercept of many technologies which are:

- conceptual design and implementation of an edge computing solution
- installation and configuration of a Kubernetes cluster with the K3S distribution
- machine learning and object detection for image classification
- development of a web application as a front-end

In the beginning, we will briefly describe the project, the resulting requirements, and the designed system architecture. The architecture is kept compact in the introduction and addressed in the dedicated sections in more detail. Hereafter the distribution of the team members into subgroups is presented.

1.1 Project Scope and System Architecture

The scope of the project is the development of an edge computing solution for the automatic detection of pests as already mentioned. Each group has received the following hardware to complete the project:

- 1 Raspberry Pi 4 Model B with
- 1 Samsung Pro Plus 32GB MicroSD memory card
- 4 Raspberry Pi 3 Model B
- 4 Samsung Evo Plus 32GB MicroSD memory card
- 2 MicroSD standard SD card adapter
- 5 CoolReall Micro USB Cable
- 1 Raspberry Pi power adapter

- 1 Raspberry Pi case
- 22 bolts for Raspberry Pi construction
- Anker PowerPort 10 (with 10 USB ports)
- 1 TP-Link 5-port gigabit Desktop Switch
- 1 Raspberry Pi camera module V2
- 5 LAN cable (1m)

With the hardware given, the overall architectural design was indirectly defined. The intended use of the Raspberry Pi 4 and the camera module was to detect rats by classifying the pictures of the camera. If the classifier detects a rat, the image and related metadata should be sent to the back-end. The term back-end is broad and a temporary description since we have not introduced its meaning yet. The four Raspberry Pi 3 create a Kubernetes Cluster with k3s distribution. The cluster should store the sent images and metadata and display them. Therefore, we have made use of a web application that is containerized and runs on the cluster. This represents our back-end. The availability of the network at the university was also considered. To present the whole system, we had to move the cluster. Therefore, we have created our own network and set the system up in it. Thus we can move the system and still control the IP addresses. The architecture is given in figure 1.1:

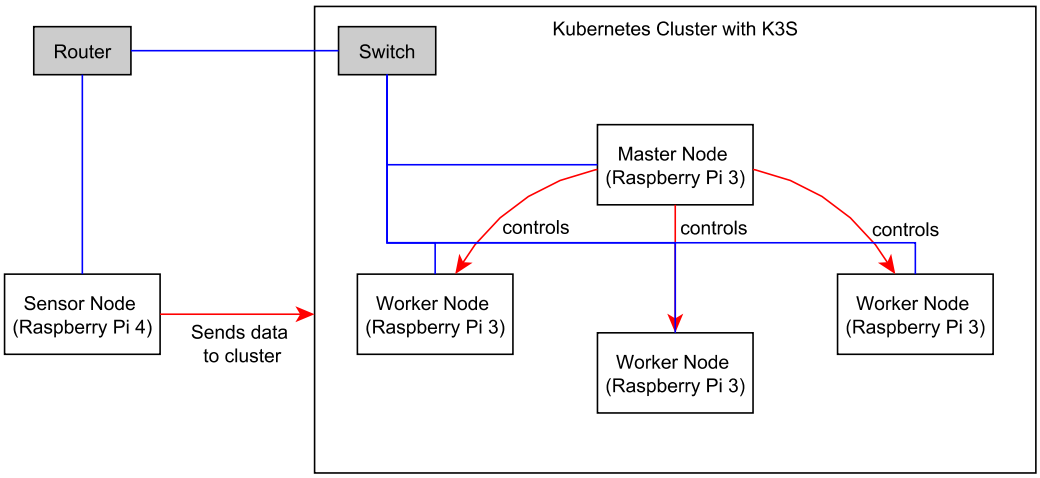


Figure 1.1: Broad Architecture of Persistent Storage with NFS-Server

1.2 Distribution of Team Members

The task distribution among the team members can be viewed in table 1.1.

Group	**Name**	**Tasks**
Cluster	Henry Kniss	Research and work on cluster setup/K3S/MinIO/persistant storage
Cluster	Christopher Unkart	Virtualization of cluster through virtual machines
Cluster	Marc Kaiser	K3S cluster setup, configuration of persistent storage and pods
Web-App	Frederik Kliemt	Development of Web-App and integration on cluster
Sensor-Node	Frederik Kliemt	Development of Sensor-Node application, Docker
Sensor-Node	Fabian Hauck	Construction/setup sensor node, camera, Telegram Bot, Docker
Sensor-Node	Lucas Merkert	YOLOv5 setup/training, integrate model into sensor node

Table 1.1: Task distribution of team members

2 | Sensor Node

This section covers the node with the sensor node. Section 2.1 covers the hardware and software set up of the Raspberry Pi 4, while section 2.2 deals with the camera. In section 2.3, the object detection of the recorded pictures is covered. Section 2.4 is about the notification of the detected rats. The sensor node chapter closes with the containerization of the code in section 2.5.

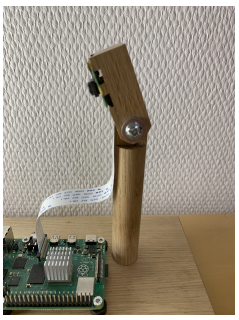
2.1 Setup

The core part of the sensor node consists of a Raspberry Pi 4 Model B including a 32GB microSD card. Raspberry Pi OS Bullseye 64-bit is written onto that card by inserting the card using the Raspberry Pi Imager application. Furthermore, a power supply, as well as a LAN connection, is required. An additional micro HDMI cable, including a monitor, were used to display the output of the sensor node. The usage of a GUI instead of SSH access was possible due to the increased power of the latest Raspberry Pi generation and improved the controlling and visualization of the results. Furthermore, all dependencies of the camera stack have been installed this way. For picture recording, the Raspberry Pi camera module V2 was provided. The mounting of the camera module has the following requirements:

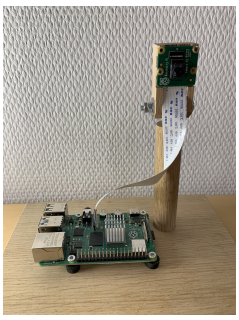
- Blur-free images
- Constant camera frames without unintended changes between sequential pictures
- Ability to easily adjust frame
- Safe and comfortable operation
- All needed connectors have to remain directly accessible

To ensure the recording of blur-free and constant images, the camera module was mounted with 2mm thick screws into a piece of wood which has to be grooved due to the space requirements of the back side of the camera module as well as the ribbon cable. The camera mounting is screwed loosely into another woodblock, enabling a pitch angle to adjust the frame. The other woodblock, in turn, is put into a hole of a wooden board drilled by a Forstner bit which enables a yaw angle. The roll angle

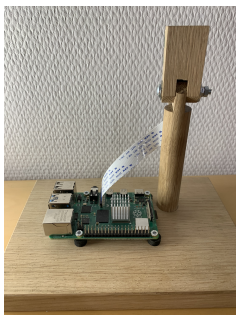
(a) Pitch angle backwards



(b) Pitch angle forwards



(c) Yaw angle left



(d) Yaw angle right

Figure 2.1: Euler angles of camera

can simply be adjusted by turning the construction. The adjustability of all three Euler angles makes the construction flexible and easy to handle as figure 2.1 shows.

The Raspberry Pi is screwed into that wooden board with two washer rubber at each screw. This way, the micro SD card at the bottom can still be accessed. The mounting enables a safe and comfortable operation since no permanent touching and holding of the Raspberry Pi, the ribbon cable, and the camera module is needed. At the same time, all connectors remain accessible.
All parts are cut by a panel saw and afterwards flattened to the optimal size by a jointer and thickness planer. A smooth surface is crucial for comfortable handling. Therefore, an edger and an edge router has been used for the edges. The bigger surfaces have been flatted with a rotary sander, while the smaller parts have been flatted with sandpaper.
Picture 2.2 shows the construction of the sensor node.

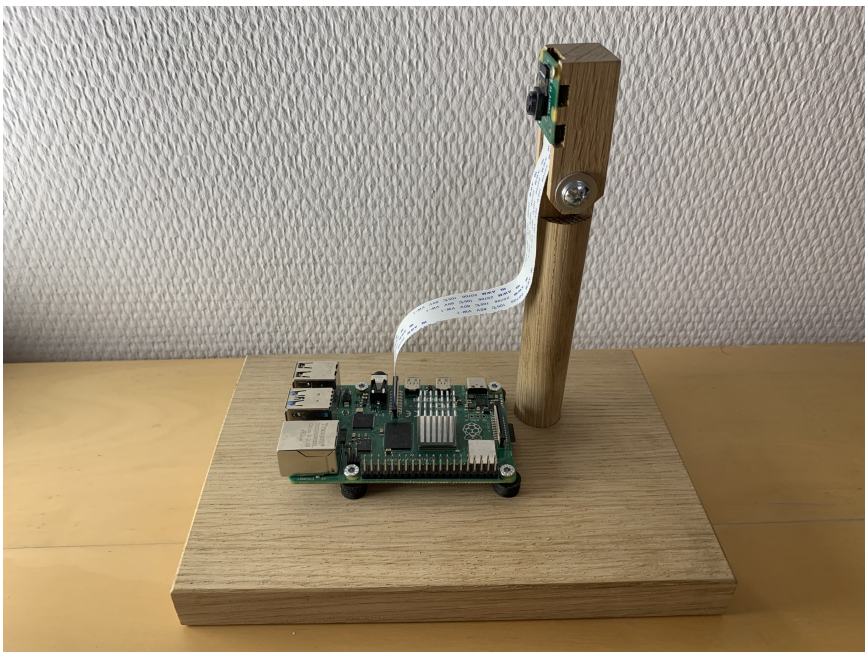


Figure 2.2: Construction sensor node

2.2 Camera

The camera module is connected to the Raspberry Pi via the camera serial interface (CSI). The camera lens has to be turned around to modify its sharpness. Regarding the software, the Python camera library *Picamera2* [11] is preinstalled on Raspberry Pi OS. *Picamera2* is the replacement for *Picamera* and build on top of *libcamera*. *Libcamera* is an open-source tool for camera operations on Linux. *Picamera*, on the other hand, is a Python library specified for Raspberry Pi and is said to be easy to use with direct access at higher level, according to the information of the manufacturer. The developer aims to combine the specialization to the Raspberry Pi of *Picamera* with the open-source approach of libcamera in *Picamera2* [10].

The camera of the sensor is accessed via a Python script that imports *Picamera2*, *libcamera*, and the time package. *Picamera2* provides several options to configure and control the camera. All possible modification options have been checked. However, since there are just a few special requirements, only the image resolution, as well as the image format, are modified to quadratic frames of size 2464×2464. The script creates two configurations, one for file saving and the other for a preview window. Besides the displaying of the preview window, a picture in JPEG format is recorded and saved every two seconds. The object detection in section 2.3 accesses this file.

2.3 Object Detection

To detect rats on recorded pictures, an object detection framework is needed. In this project, YOLOv5 [15] was chosen. YOLOv5 is a commonly used framework to detect objects in pictures and videos. As a base, YOLOv5 uses Pytorch as the neural network library. There are several versions of YOLO (v1 - v8 currently) existing. After research, Yolov5 was chosen because it has been used for rat detection before [6].

2.3.1 Training setup

For a setup with CUDA, follow this tutorial [16]. Furthermore, a set of labeled training data is needed. For the training data, the provided data set of ~ 500 labeled rat pictures was used. Another ~ 1000 pictures were taken of privately owned rats. The complete data set was uploaded to Roboflow [14], which is a tool helping to label, divide, and distribute a data set. With Roboflow, unlabeled pictures can be automatically assigned to all members equally distributed. The data set was then divided into training, validation, and test subsets with a ratio of $70\% - 20\% - 10\%$. In Roboflow, the data set can be downloaded with the specific setting for YOLOv5. The data set was stored in the YOLOv5 folder.

2.3.2 Training

To train the YOLOv5 convolutional neural network, navigate to the YOLOv5 folder and execute the following command:

```
python train.py --img 640 --batch 10 --epochs 200 --data path/to/dataset.yaml --weights yolov5l.pt
```

Depending on the machine used, this may take a long time. The options that can be chosen for the training process are:

- –img: quadratic image size in pixels
- –batch: training data used per training epoch
- –epoch: maximum number of training steps
- –data: dataset.yaml, that comes with the downloaded data set
- –weights: pre-trained weights from Yolo

For the last option, there are several pre-trained weights to choose from. Picture 2.3 shows a comparison of the pre-trained weights. After completing the training, YOLOv5 provides an output folder at

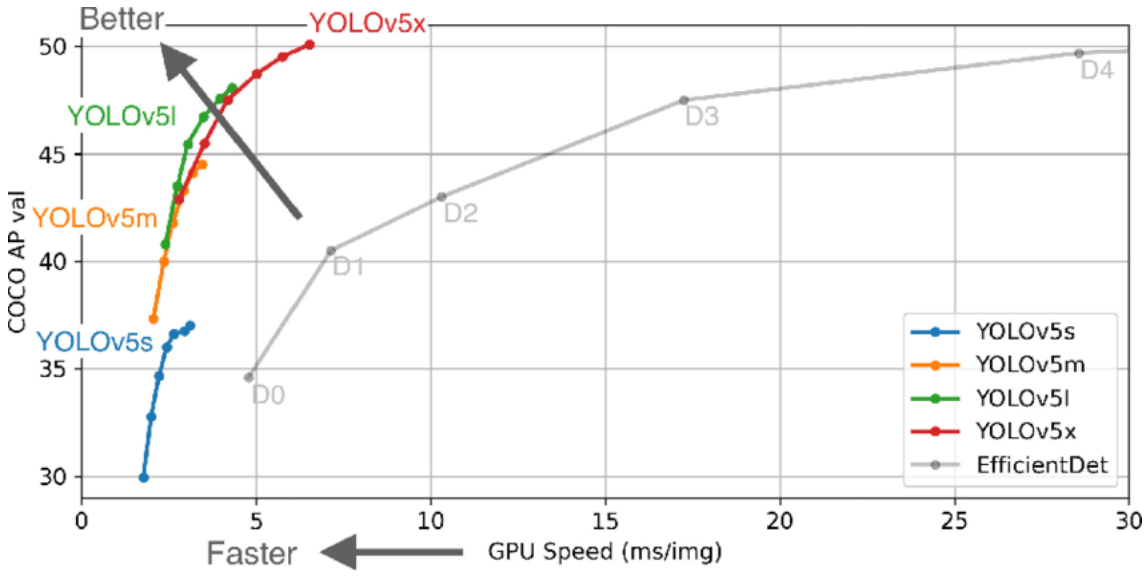


Figure 2.3: Pretrained YOLOv5 weights comparison[15]

"/path/to/yolov5/runs/train/expXY/". The folder contains the weights, statistics, and a sample of the detection on the validation and training subset. For this project, the weights "/weights/best.pt" were chosen. These need to be copied into the "/SensorNode/weights/" folder on the sensor node.

The statistical results can be seen in Picture 2.4. As shown in the graphs there is a differentiation between the training and the validation. The training is done in step that are called epochs. The difference between training and validation is that per epoch the training subset is used to train the

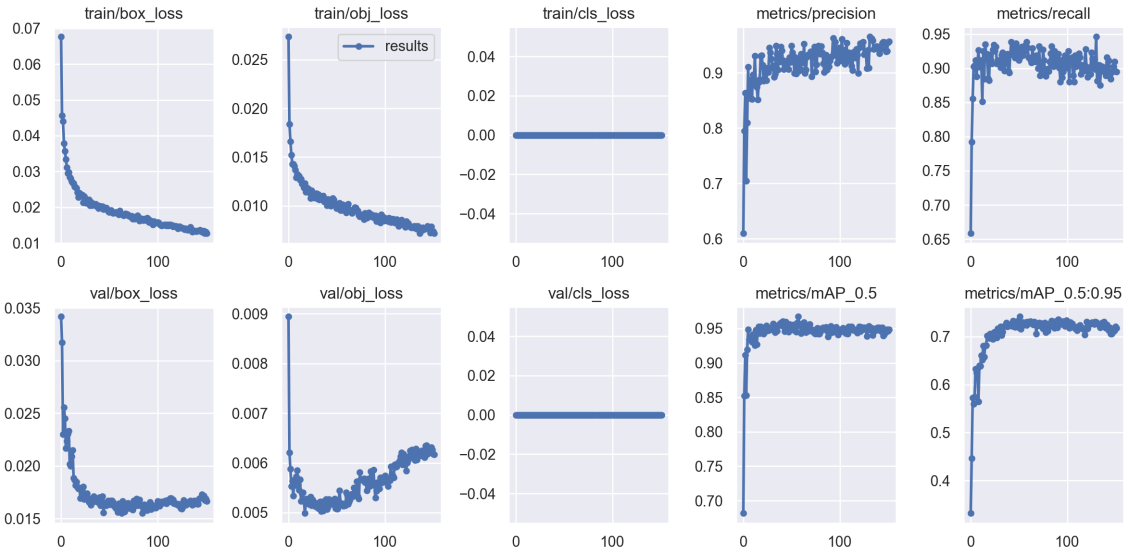


Figure 2.4: Statistical Results of the Yolov5 Model Training

model. After each epoch, the model is validated using the validation set. This ensures, that as soon as the error of the validation set increases and the error of the training subset decreases, the training is stopped to prevent overfitting. Box loss represents how well the algorithm can locate the center of an object and how well the predicted bounding box covers an object. For the training as well as the validation, the box loss decreases continuously. Object loss represents how well the algorithm predicts a region that contains an object. Here the object loss for the training decreases while the validation object loss increases. This indicates slight overfitting, which could be solved by providing more data. These pictures should have rats in many different locations to teach the algorithm that a rat can be anywhere in the picture. The class loss represents how well the algorithm predicts the correct class. This loss is zero because there is only one class to detect so there cannot be a wrong predicted class. The precision is calculated as: $precision = true\ predictions/total\ predictions$, the recall is calculated as: $recall = true\ predictions/(true\ predictions + false\ negative\ predictions)$. The mean average precision (mAP) is calculated for a confidence threshold of < 0.5 and > 0.5 and < 0.95. Both precision and recall increase during the training as well as the mAP0.5 and mAP0.5-0.95. Overall the metrics are standard for model training. After testing the model, there still have been many false predictions. The confidence threshold was set to 70% to counteract these false predictions. Additionally, more training data would be needed to train a better model. Since this would take a lot of time and effort to see a significant change, training a better model is out of the scope of this project.

2.4 Notification System

Outputting the recognized rats is crucial for fulfilling the system requirements. While the cluster part in chapter 3 provides a web application (chapter 4), an additional system with direct notifications and high usability is advantageous. A Telegram Bot has been chosen as notification system since Telegram is a quite popular and intuitive application with a programmer-friendly Bot API. The Telegram Bot

aims to provide a direct and intuitive user notification system. The Bot written in Python is integrated into the sensor node. The user has to subscribe to the Telegram channel "RatDetectionBot". Every new user is added automatically and will receive pictures of detected rats, including information about the number of rats as well as the corresponding confidences. Picture 2.5 shows an exemplary notification with two rats.

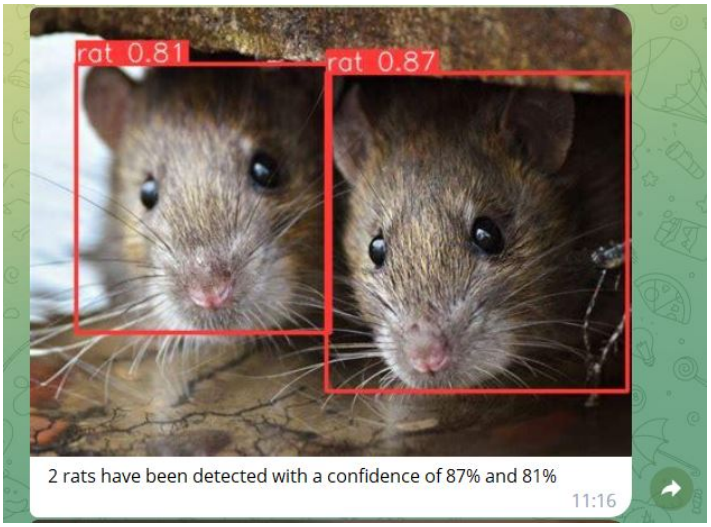


Figure 2.5: Notification of the Telegram Bot

Furthermore, every random input message activates a personalized menu with two selectable options. The first one is toggling notifications on and off. This way, the user can activate and deactivate the automatized sending of rats. The second option is the modification of the notification interval. To prevent the permanent sending of rat notifications if a rat is recognized over a long period, the sending interval between two pictures can be typed in. The Bot provides an accompanying input string control. The alert is turned on by default, and the notification interval is 600 seconds (10 minutes). Every user has individual settings. Figure 2.6 shows the menu of the Telegram Bot.

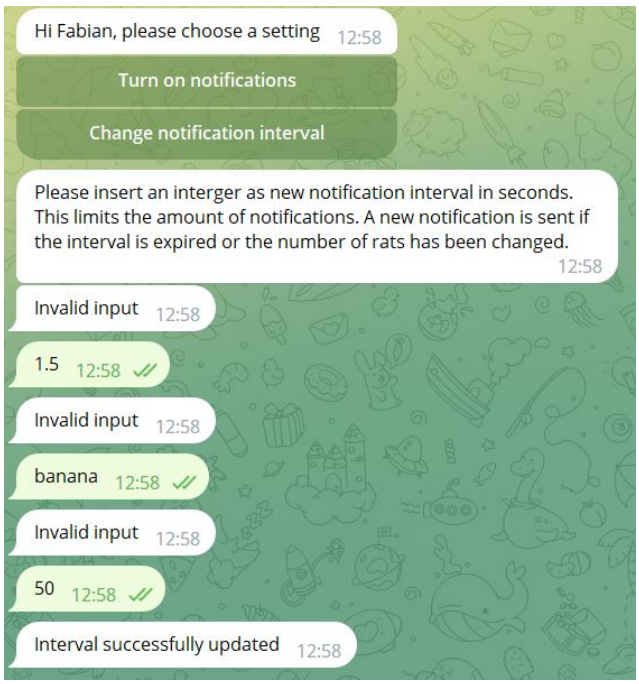


Figure 2.6: Menu of the Telegram Bot

2.5 Containerization

Docker has been used as tool for container virtualization. The object detection runs inside a Docker container based on an the YOLOv5 image. The standard YOLOv5 image is exclusively designed for amd64 architecture. This image works on Raspberry Pi OS virtual machine if the CPU has amd64 architecture but not on Raspberry Pi 4 CPU with arm64 architecture. Therefore, an image with an arm64 tag from YOLOv5 version 6.2 is used. It has been shown that the latest arm64 tag is not working correctly.

Furthermore, there have been problems with the Picamera2 library in Docker. First of all, YOLOv5 requires Python with version ≤ 3.8, and Picamera2 needs Python ≥ 3.9. Hence two different Docker containers are needed. While the Docker container for YOLOv5 is working well, the Picamera2 container has several problems. Since there is no official Docker container for Picamera2, this image has to be built from scratch based on a Python image. All attempts to build such a container have failed at the step of downloading Picamera2. Even after using a Python image with Bullseye OS and adding "contrib" and "non-free" repositories to the accessible packages. A working Picamera2 image might be available as soon as there comes up a final release instead of the current beta release. In this project, this issue was solved by running the camera script outside of Docker and saving the results in a folder that is accessed by the YOLOv5 container through a bind mount. To integrate the model in the sensor node python script, follow the tutorial on GitHub[5].

3 | Cluster

This part of the documentation explains how to install the Kubernetes distribution K3S on a set of Raspberry Pi 3B or virtual machines and how to configure the cluster for the project. It also describes how to set up virtual machines and how to run K3S without needing additional hardware. The K3S distribution can be found here [2].

3.1 Hardware Requirements

- At least 2 Raspberry Pi 3B
- An external SSD hard drive
- Administrator access to the router of your network
- LAN- and power cables for the Raspberries

3.2 Preparations

First, install the Raspberry Pi Imager [12] on your PC. Other applications do also work to flash the images of the operating systems (OS) on the micro-SD memory cards, but this particular application has two advantages. The imager has the Raspberry Pi operating systems already available. It also has an "advance settings" menu, which can assign a hostname, enable SSH, assign a username and a password, and set up the Wi-Fi connection to your network. This means that you only have to plug the micro SD card into the Raspberry after flashing it. The OS we are using is the Raspberry PI OS Lite (32-BIT). The main reason for using a 32-Bit architecture is to reduce the overhead. Raspberry Pis have low computing power compared to the servers used in a data center. It seems reasonable to give the devices host names equivalent to their purpose, for example, "masternodeX" or "workernodeX". This setup was built with one master node and three worker nodes. Therefore, we named the devices:

- masternode1
- workernode1
- workernode2

- workernode3

In the following sections, the "masternode1" is just referenced by "masternode". Plug in the flashed micro-SD cards, connect the Raspberries via LAN to your network, and activate them. Now check if the devices are in your network. If the devices are in the network, continue with the installation.

3.3 Installation of the Virtual Machines

Instead of installing Raspberry Pi OS on a Raspberry Pi, it can also be installed on a virtual machine. This replaces non-existing hardware. First, an application that supports creating and running virtual machines is necessary. VirtualBox was installed and used for creating and running the virtual machines in this case. In the chosen program, a new machine needs to be created. For the creation of a new machine, an ISO image is necessary. Through the ISO image, a new operating system can be installed.
Since we want to use the virtual machines to simulate a Kubernetes cluster that runs on multiple Raspberry Pis, the Raspberry Pi OS Desktop (32-bit) ISO image is chosen to create the virtual machine. The other versions did not include an ISO image and were unsuitable for installing the virtual machine. After the ISO image has been downloaded, the virtual machine can be created. By creating the virtual machine, the name, the folder of the installation, and the ISO Image that should be used to install the virtual machine have to be specified. This is why we have downloaded the ISO image previously before creating the machine. When everything is correctly selected, we can continue. We chose to skip the unsupervised installation of the machine. After continuing, the power of the virtual machine can be adjusted. The number of processors and the Random Access Memory (RAM) of the machine can be chosen. To better simulate the Raspberry Pis, lower values are chosen here. A Raspberry Pi 3 usually has 1024 MB RAM, while a Raspberry Pi 4 can have more. We initially started with 2048 MB RAM and 2 cores, which can be adjusted whenever necessary.

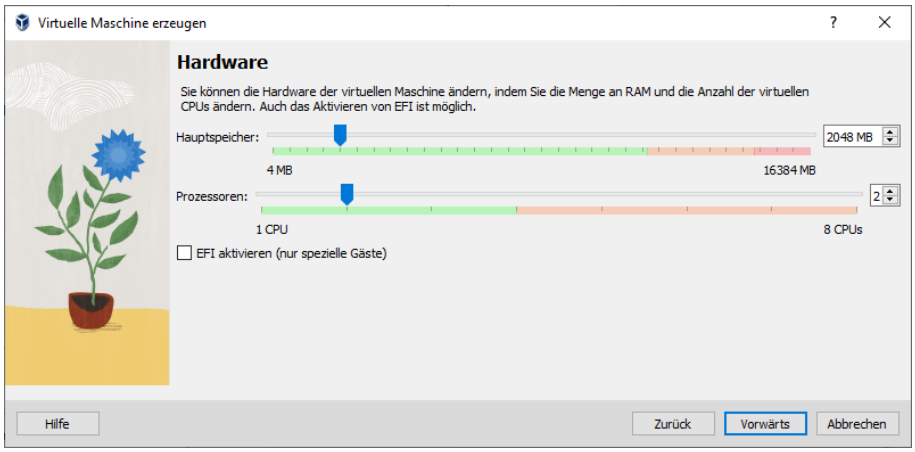


Figure 3.1: Limiting the power of the virtual machine in VirtualBox

The virtual disk is created with 20 GB of space. Hereafter a summary of the settings is shown, and we can finish the setup of the virtual machine. Thus, the machine can be started by selecting it in

the menu and pressing the start button in VirtualBox.

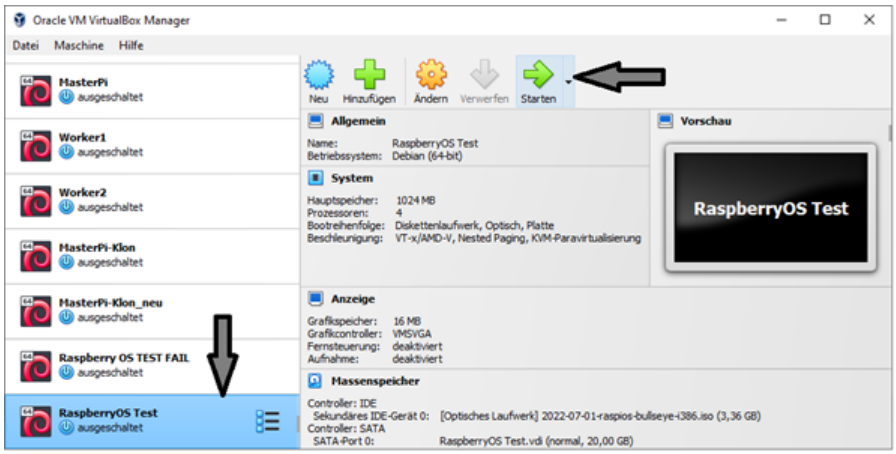


Figure 3.2: Starting the freshly created virtual machine

After starting the virtual machine, it is booted from the previously selected ISO image, and the Raspberry Pi OS has to be installed. For this, the install option has to be selected manually. This is different from installing Raspberry Pi OS through the imager since no further installation steps were necessary after flashing Raspberry Pi OS onto the SD card.

The next step is to select the language of the Raspberry Pi OS. For this machine, German was chosen. After selecting the language, the disk has to be partitioned. Because of the setup of the virtual disk of the virtual machine, we only have one option here and, thus, not much to choose from. As the partitioning method, guided partitioning of the entire disk is chosen and for the partitioning scheme, storing all files in one partition is chosen. After confirming, that partitioning is finished, the changes are written to the disk, and the system starts installing. If chosen, the GRUB bootloader can be installed. After this, the installation is finished, and booting into the virtual machine is possible. The freshly installed virtual machine is cloned and saved as a backup. Therefore if the cluster machines break, a rollback is easily possible. We can also clone the machine multiple times, dependent on how the cluster is built. For our case, we set up the cluster with one master node and three worker nodes, and thus need four virtual machines to simulate that. The installation and configuration process of K3S is described below. Machines can be removed or added by simply deleting or cloning the virtual machine, which makes the virtual setup easily scalable.

3.4 Installation of a K3S-Cluster

A large part of this section originates from [9]. The next steps describe the installation of the cluster. Execute the following command on each of the Raspberries to update and upgrade the software:

```
1     sudo apt update && sudo apt upgrade -y
```

This can be done simultaneously to reduce the waiting period. After that, execute the following command on the master node:

```
    curl -sfL https://get.k3s.io | sh -
```

During the installation, an error will occur, but the installation itself will not be interrupted. Thus execute:

```
    sudo nano /boot/cmdline.txt
```

This opens the "cmdline.txt" document in a text editor. The document has only one line and must be extended by the following line:

```
cgroup_memory=1 cgroup_enable=memory
```

It is essential not to leave the first line and accidentally insert a line break. Before the master node is rebooted, we must get the node-token. Hence execute:

```
    sudo cat /var/lib/rancher/k3s/server/node-token
```

This string is important for the installation of the worker nodes. It must be temporarily saved somewhere. After that, reboot the master node and continue installing the worker nodes.

```
    sudo reboot
```

The installation of the worker nodes is almost identical to the installation of the master node. Install K3S with the following command and insert the master node IP address and token in the marked fields:

```
    curl -sfL https://get.k3s.io | K3S_URL=https://<masternode_IP_Address
   >:6443 K3S_TOKEN=<masternode_token> sh -
```

Again, the installation will yield an error, but the procedure will not be interrupted. Execute the same command used at the master node to correct the error. Reboot the worker node after that:

```
    sudo reboot
```

Repeat this installation process for every worker node. If the installation has been successful can be checked by executing:

```
    sudo k3s kubectl get nodes
```

Should all nodes have the status "Ready", we can continue with the configuration.

3.5 Configuration of the K3S-Cluster

The configuration of the cluster is based on the requirements of the project. Another aspect we have considered is the limits of the hardware at hand. The cluster must maintain persistent storage and display the classification results from the sensor node in a WebApp. The hardware limitations are relevant in a later stage of the documentation. The configuration is applied to the cluster in the form

of a YAML manifest via the Kubernetes API. At the start, we will explain our decisions which lead to the final cluster architecture and, hereafter, the YAML manifest itself.

3.5.1 Theoretical Decisions and Application

By default, each node only accesses its own file system. There are several possibilities for creating persistent storage for the Kubernetes cluster. The first possibility is to label the cluster nodes and specifically define the node on which each application must run. This procedure seems to be technically possible, but from our point of view, it represents the exact opposite of the intended use of a Kubernetes cluster. The second option we considered was the object storage application MinIO [7]. The last possibility we have considered was an NFS-Server hosted on the cluster. We have split up the hardware to determine which of the two options works best. After testing, we concluded that MinIO created too much overhead, and therefore, the application kept crashing the cluster nodes. Also, during the MinIO setup, we ran into many issues concerning the pod in which it was supposed to run, while the NFS-Server worked without a problem. Thus, our decision was mostly guided by pragmatism and led us to the use of the NFS-Server instead of implementing the theoretically well-suited MinIO. We have used of an external SSD hard drive to enhance the available storage. The goal of this enhancement was to reduce the number of reads/writes on the micro-SD card. We plugged the SSD into the master node and bound the pod of the NFS-Server to the node with a label.

At first glance, this method seems to contradict our previous statement about the intended use of a Kubernetes cluster. However, here we argue with the hardware limitations. A productive cluster could use Network Attacked Storage, or one could configure predefined solutions for persistent storage. To the best of our judgment, the Raspberries used here seem to lack the capabilities to run MinIO and our applications in a manner suited for this project. Since we only have one master node in our architecture, we are not creating more single points of failure.

It has to be stated that our decision between the NFS-Server and MinIO might have been biased. During the project (after our testing period), we swapped the micro-SD cards of the MinIO nodes for newer ones. This has led to a notable increase in performance. At this point, we have already built on our earlier decision, and a change was not easily possible anymore. Hence, we stuck to the NFS-Server.

To set up the NFS-Server, the master node has to be labeled with the command:

```
sudo kubectl label node <masternode_hostname> SSD=enabled
```

Since we plugged the external SSD in via USB, we wanted to ensure it is always booted into a defined directory. Hence we extracted the UUID:

```
sudo blkid
```

After that, we inserted the following string into the file-system table /etc/fstab

```
UUID=<UUID_of_SSD>   /media/usb   ntfs   auto,nofail,sync,users,rw   0
0
```

With this configuration, the SSD is permanently mounted into a defined directory after a reboot. After that we copy the manifest directory to the master node and apply it to the cluster with:

```
sudo kubectl apply -f ./manifest
```

To get the running pods, we can use the command:

```
sudo kubectl get pods -n work-space
```

Since we created our own namespace, we have to extend the commands by "-n work-space". To get access to the MariaDB, the following commands must be executed.

```
sudo kubectl exec -it <Containername> -n work-space -- /bin/bash
```

This opens the bash of the pod with the MariaDB. To get the MariaDB console, type:

```
mariadb -p
```

Enter the password as requested. The password of the MariaDB is currently "password". It is saved in the "secret.yaml" file and is base64 encoded. Do not use this password in production! When the console of the database is opened, execute the SQL script stored in "Rat-Detector/RasPiCluster/WebApp/data/sql_scripts"

3.5.2 Kubernetes Manifest

The "manifest" directory has several files that are applied to the cluster with the kubectl API. These files are (mostly) named by their Kubernetes API kinds. It must be stated that the "deployments.yaml" file also contains the "namespace" kind. Thus the directory contains:

- deployments.yaml
- persistent_volume_claims.yaml
- persistent_volumes.yaml
- secrets.yaml
- services.yaml

The contents of these files originate from multiple sources. We will not explain every aspect of the configuration but rather point out significant aspects of the architecture and setup. An important source for our configuration was the Kubernetes documentation [3]. We also used additional sources, which will be mentioned in the specific sections.

For the configuration of the NFS-Server setup, we also used [13]. The NFS-Server makes use of four Kubernetes API kinds. These are "Service", a "Deployment", a "PersistentVolume" (PV) and a "PersistentVolumeClaim" (PVC). Figure 3.3 gives a broad overview of our architecture for persistent storage.

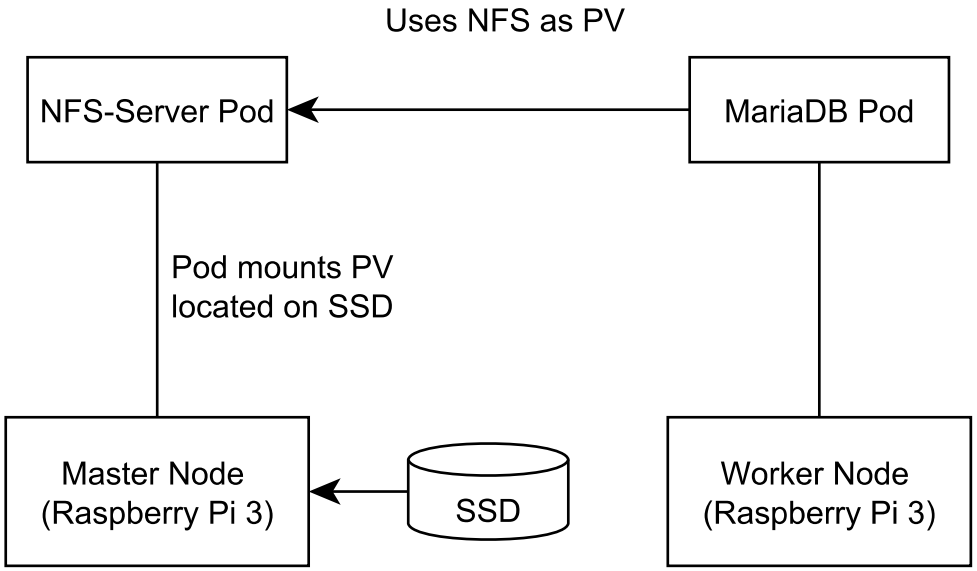


Figure 3.3: Architecture of Persistent Storage with NFS-Server

An important aspect of this configuration is the setup of persistent storage. At the start, we created a PV. This represents a piece of storage that has been configured based on the given capacities. The "accessMode" defines the type of access and the number of nodes that can mount it. The value "ReadWriteOnce" implies that only one node can mount the volume, which has read and write access. Under the "nodeSelectorTerms" we define the label, which is a selection criterion for the PV. Only one node has the external SSD, and with this flag we can determine to which node the PV is assigned. The "path" attribute is mentioned because this path is mounted by the SSD. Hence the data of the NFS-Server is stored on the SSD. A PVC is a counterpart to the PV and thus necessary for our setup. The PVC represents a request for storage. The specifications of the request are defined in the "persistent_volume_claims.yaml". In our case, it just contains the capacity. If the PV and the PVC are bound together, they represent a one-to-one mapping.

Deployments manage the "desired state" for pods. They are specific to one particular type of image. If the current state is not equal to the desired state, Kubernetes changes the current state until the desired state is reached. In this configuration, one can specify network settings or the number of simultaneously active pods. In "deployments.yaml" the "nfs-server" contains three important elements. The first element is the "nodeSelector". Its purpose is equal to the "nodeSelectorTerm" of the PV. The second and third elements are the "volumes" and "volumeMounts" attributes. In the "volumes" attribute, we defined the PVC as storage capacity for this deployment. In "volumeMounts" we mounted the "volume" against the path "/exports". This means we indirectly mounted the directory /media/usb/share (defined when setting up the SSD and the PV) on the SSD into the NFS-Server.

The last important element is the service. Kubernetes distributes IP addresses dynamically. In this specific case, this behavior is not desired. Hence we have given the pod a static IP address.

The remaining configuration is used to deploy the WebApp. It is composed of three parts: the MariaDB database, the WebApp-Server, and the WebApp-Client. For this configuration, we have additionally used an example from the Kubernetes documentation [4]

WebApp uses MariaDB as a database. The reason for this is the intercept out of the following properties. MariaDB is available on a 32-Bit architecture, it is relational and can be used to store images, and it has a predefined connector to the nodeJS framework.

For MariaDB also exists a deployment. Similar to the NFS, we have defined a "volume" and "volumeMounts". This time the "volume" was the NFS-Server which we have mounted into the path where the MariaDB saves the data ("/var/lib/mysql"). This configuration displays the reason why we have chosen a static IP addresses when configuring the NFS-Server. It allows us to specify the IP address with certainty. A unique configuration in this setup is the "secrets.yaml" file. Under the "secretKeyRef" tag in the deployment, Kubernetes uses an encrypted string, which is the password to the database itself. Secrets are used to divide confidential or sensitive data from the application code. Similar to the NFS-Server, we have also created a service that defines a static IP address for the database.

The remaining two deployments and services are used for the WebApp. The images of the WebApp are stored in the DockerHub. There were mainly two reasons to store the images there, the ease of use and the wide range of people using it. The deployments ensure that one pod of the server- and client part is always up and running. The remaining services are from the kind "LoadBalancer". This service provides an IP address for the pods that can be accessed from outside the cluster. We forwarded ports of the pods to two distinct "nodeports". The "nodeports" have, by default, a range of 30000-32767, hence the high "nodeports". The IP address of the MariaDB is statically defined before the WebApp Image is compiled and pushed to DockerHub. Therefore, in a new setup, the images of the WebApp have to be recompiled with changed IP addresses.

4 | WebApp

In order to be able to handle all rat detections and make them available on a web interface for a user, our team decided to develop a Web-App. We split the tasks that should be managed by the Web-App, to obtain a better overview over the jobs at hand.

- The Web-App should handle the traffic of detections from the sensor node to the database and save them there.
- It should be able to fetch the saved detections from the database and route them to the front-end.
- The front-end should be able to display the images of the detected rats plus metadata.

4.1 Design and functionality

4.1.1 Architecture

See figure 4.1

4.1.2 Technologies

For the front-end, Vue.js, in combination with Vuetify for quick and easy styling was used. These frameworks make it possible to quickly and easily design a dynamic, modern-looking Web user interface. Express was used for the back-end to build an easy and robust RESTful API quickly. As Database was MariaDB used. Since there was the need for a local storage instead of a cloud storage on the internet like it would be the case with, for example, MongoDB. This was the case because the project was supposed to work without an internet connection.

4.1.3 Front-end

The Web-App front-end was designed for intuitive and easy use. The 30 latest rat detections will be displayed with the image of the detection and the time-stamp when the detection occurred. To make it obvious that the images are clickable and entail more information, the mouse cursor switches from a normal mouse icon to the pointer icon, and the images have a hover effect that makes the

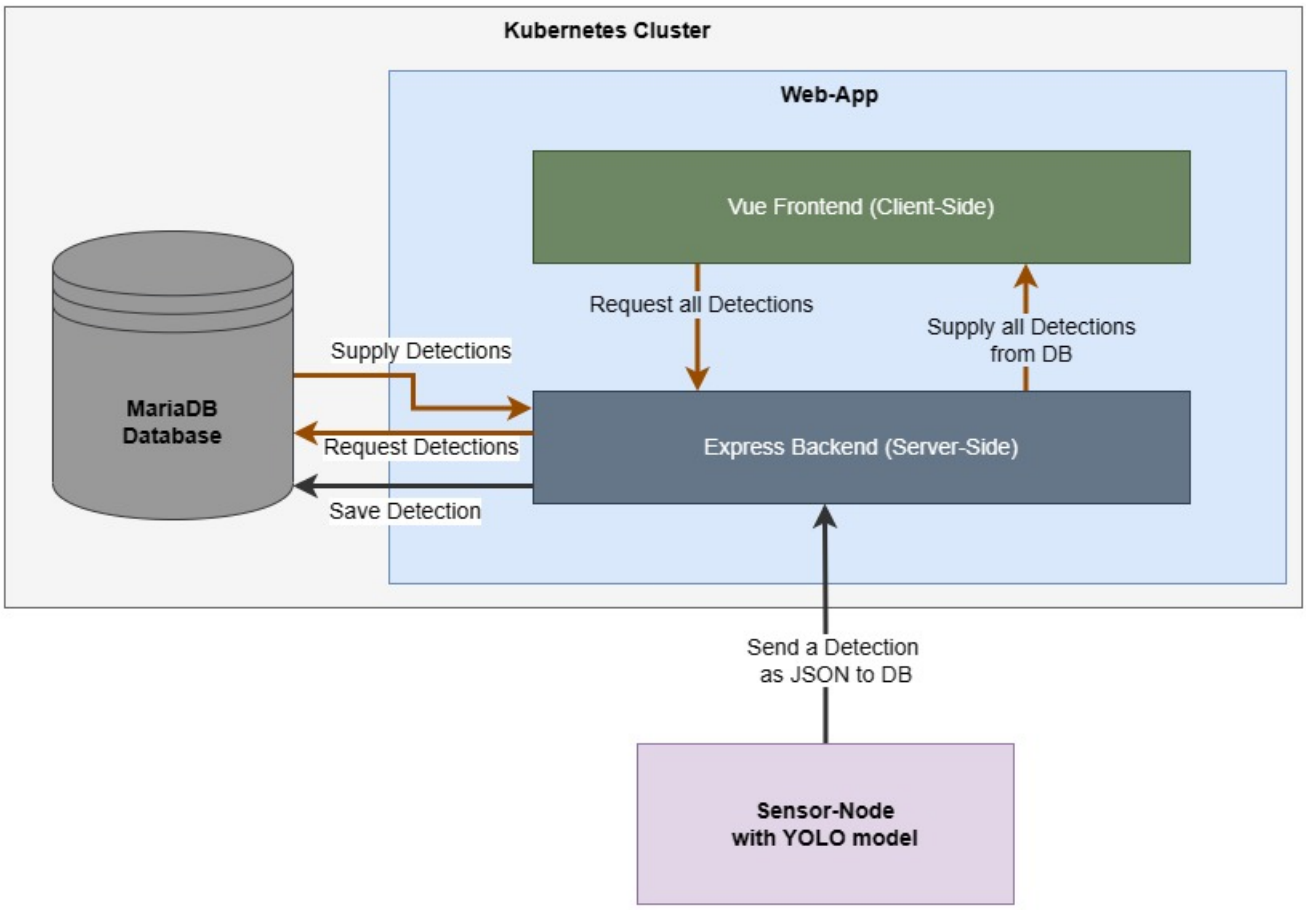


Figure 4.1: Architecture of the Web-App

image appear elevated for a more intuitive user experience. More information about the detection will be visible when the image is clicked. The additional information is the number of detected rats and the according confidence. In order to receive the detections with their according metadata, a detectionService.js was implemented that makes the necessary calls to the back-end and the processes and displays the response.

4.1.4 Back-end

The back-end was responsible for routing the data from the sensor node to the database and from the database to the front-end for it to be displayed there. In order to be able to handle those tasks, the back-end offers an API. There are essentially two endpoints that can be used and that are reachable under the configured back-end address, plus "api/detections". So if the Web-App is running on a local machine, the address could look like this: "http://localhost:5000/api/detections". The first functionality realized in the API is the data flow from the sensor node over the back-end to the database. If the sensor node detects a rat, it can send a REST post-request to the API address, where the body contains data needed to save a rat detection correctly. To be precise, the back-end expects to receive JSON data that sends the base64 encoded JPG of the detection. Additionally, the number of detected rats and their confidence levels must be sent. The datetime of the detection does not have to be sent, since the

timestamp will be created as soon as the detection arrives in the back-end. An example requests body could look something like this:

```
{
    "image": '/9j/2wBDAA0JCgsKCA0LCgsODg...',
    "NumberOfRats": 3,
    "confidences": '88.5, 90.3, 78.0'
}
```

After the request is received and processed correctly, the back-end will send a response with the HTTP status 201 and a message informs the user that the operation was successful. In order to save the information that the back-end received from the sensor node in the database, the MariaDB Node.js Connector was used. This library is a native JavaScript driver that allows it to send SQL-Queries within JavaScript and use placeholder-parameter-loading to insert variables into the queries. The initialization of the connector can be found "RasPiCluster/WebApp/server/DBConnector/db_connector.js"

The second functionality that is implemented by the API is fetching data from the database and the return the data in a response body. So, if a get-request is sent to the API, the back-end will use the MariaDB Node.js Connector to send a query to the database in order to get all the detections from the database. Those detections will then be sent back as JSON in the response body.

A diagram that illustrates these workflows can be found under 4.1

4.1.5 Database

The requirement for the database was to save the image of the rat detections with additional metadata. The metadata chosen to be of interest were the number of rats, the confidence levels with which they were detected, and the time when they were detected. In order to realize such a database, an initialization script was provided, which can be found under "RasPiCluster/WebApp/data/sql_scripts/init_database_and_table.sql". This script gets executed every time a new docker container for the database is created via the docker-compose.yml. This script ensures that the tables in the database are correctly built and initialized if they do not already exist.

4.2 Deployment

4.2.1 Docker Deployment

For easy and quick system-independent deployment, we provided Docker-Compose and Docker files. This makes it possible to start the Web-App on any machine that can run docker with one easy command. A few things have to be acknowledged before trying to start the Web-App. The Docker-Compose and Docker files are programmed to work on the Kubernetes Cluster in parallel. A few things have to be altered if the Web-App is supposed to run on an amd CPU architecture. To let

them run on a local amd Windows/Linux machine, for example, the line containing the "platform" flag (line 4 and 10) in the "Rat-Detector/RasPiCluster/WebApp/docker-compose.yml"-file, has to be deleted. Furthermore, the static IP addresses configured on our cluster (see. 3.5.2) are hard-coded into our code. Two of the static IP addresses are of interest when trying to deploy the Web-App locally using docker, the one of the MariaDB database and the one of the Express Back-end. The static IP address of the MariaDB can be found in "Rat-Detector/RasPiCluster/WebApp/server/DBConnector/db_connector.js". In order to run the docker-compose.yml locally, it is needed to delete line 6 containing "host:'10.43.184.232'" and then uncomment (deletion of the hashtag) line 4 containing "host: 'mariadb'". The static IP address of the back-end is used in the front-end in the "RasPiCluster/WebApp/client/src/detectionsService.js"-file. Here it is again necessary to delete a line of code, in this case, line 3. In exchange for that, line 2 has to be uncommented (deletion of the hashtag). Now that all precautions were made in order to run the docker-compose file on a local amd Windows/Linux machine, all docker images can be built while being in the "RasPiCluster/WebApp" directory, using the command:

```
1    docker-compose build
```

Building the images for the three services may take a few minutes. However, after this process is finished successfully, the services can be started using the following command while still in the same directory as before:

```
1    docker-compose up
```

In some cases, the database does not boot up fast enough, and the server that is trying to establish a connection to the database might crash with a "timeout - Connection could not be established" error message. In this case, it is enough to wait for the boot process of the MariaDB service and then start the server service again using the following command:

```
1    docker-compose run server
```

When everything is up and running, the back-end will be reachable under "localhost:5000" and the front-end under "localhost:8080". Now the complete functionality can be accessed under the mentioned addresses. For the functionalities that are provided, read 4.1.

4.2.2 NPM Deployment

Npm is a package manager and runtime environment that can be used to run javascript applications. In order to install npm, follow this tutorial: https://docs.npmjs.com/downloading-and-installing-node-js-and-npm . In order to deploy the Web-App locally on an amd Windows/Linux machine, using the npm package manager [8], a few changes in the code have to be made, analog to the changes in 4.2.1. As already mentioned, some IP addresses were hard-coded into the source code of this project because it would have needed more time to implement a more convenient solution. The static IP address of the MariaDB can be found in "Rat-Detector/RasPiCluster/WebApp/server/DBConnector/db_connector.js". In order to run it locally, it is necessary to delete line 6 containing "host:'10.43.184.232'" and then

uncomment (deletion of the hashtag) line 5 containing "host: 'localhost'". The static IP-Address of the back-end is used in the front-end in the "RasPiCluster/WebApp/client/src/detectionsService.js"-file. Here it is again necessary to delete a line of code, in this case, line 3. In exchange for that, line 2 has to be uncommented (deletion of the hashtag). Now the services can be started. Open a terminal for each of the services, so in our case, three, and navigate to the "server", "client" and "WebApp" directories, respectively. In the terminal on the "client" directory, run the command:

```
1    npm install
```

Followed by:

```
1    npm run serve
```

The front-end will now be reachable under "localhost:8080".

In order to run the MariaDB database service, use the terminal in the "WebApp" directory. Run the following command in order to start the database:

```
1    docker-compose run mariadb
```

The database will now be reachable under "localhost:3306".

Lastly, we want to run the server service. To achieve this, use the terminal in the "server" directory and run the command

```
1    npm install
```

Followed by:

```
1    npm run dev
```

The backend will now be reachable under "localhost:5000".

5 | Summary

In this section, we will give a retrospective on the project and state challenges as well as possible enhancements and improvements. The project requirements have been met overall. Throughout the project, there were challenges and accomplishments. Those challenges have been already described in the dedicated sections in more detail. Therefore, this section represents a short summary of them.
The Picamera is a suitable camera module for the Raspberry Pi. The Picamera requires any kind of mounting to take sharp pictures. YOLOv5 is easy to install and use. A meaningful model for a specific task can be derived in a couple of hours provided the existence of a data set. A special challenge was faced when dockerizing the YOLOv5 image and especially the script managing the camera module. Regarding the improvements and enhancements on the end of the sensor node, it should be mentioned that with the existing system, it would be possible to change the detection class of the model running on the node to one of the many already learned classes of the model. Those would support the detection of other objects instead of rats. The Telegram bot is a working solution for direct notifications to the end user.
One of the biggest challenges of the cluster team was the scarcity of hardware. The necessity for all team members to possess at least two raspberries at any time has forced us to constantly exchange hardware and come up with creative solutions to keep all teammates able to work (section 3.3). Furthermore, one of the more significant problems was created through the logistics for the final presentation, which forced us to find a way to get a working internet connection during our final presentation at the university. This unique challenge was created through the security setup within the eduroam network. Concerning the improvement of the cluster itself, the most glaring weakness is the single point of failure originating from the master node being in the only controller. In addition, the master node doubles as our persistent storage, as described in section 3.3. There are several solutions for this problem besides implementing MinIO correctly. One of those includes creating a backup controller that provides redundancy by mirroring our data storage.
All in all, this project has demonstrated the application of cloud computing using rat detection.

Bibliography

[1] Christian Baun. *Cloud Computing (WS2223)*. URL: https://www.christianbaun.de/CGC2223/index.html (visited on 02/14/2023).

[2] *K3S: Lightweight Kubernetes*. URL: https://k3s.io/ (visited on 02/14/2023).

[3] *Kubernetes Documentation*. URL: https://kubernetes.io/docs/home/ (visited on 02/14/2023).

[4] *Kubernetes Documentation: Example Deploying WordPress and MySQL with Persistent Volumes*. URL: https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/ (visited on 02/14/2023).

[5] *Load YOLOv5 from PyTorch Hub*. URL: https://github.com/ultralytics/yolov5/issues/36 (visited on 02/14/2023).

[6] de Menezes et al. "Mice Tracking Using The YOLO Algorithm". In: 2019.

[7] *MinIO: Multi-Cloud Object Storage*. URL: https://min.io/ (visited on 02/14/2023).

[8] *npm Webpage*. URL: https://www.npmjs.com/ (visited on 02/14/2023).

[9] Alex Ortner. *How to build a Raspberry Pi Kubernetes Cluster with k3s*. URL: https://medium.com/thinkport/how-to-build-a-raspberry-pi-kubernetes-cluster-with-k3s-76224788576c (visited on 02/14/2023).

[10] *Picamera2 Manual*. URL: https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf2 (visited on 02/16/2023).

[11] *Picamera2 Repository*. URL: https://github.com/raspberrypi/picamera2 (visited on 02/16/2023).

[12] *Raspberry Pi OS*. URL: https://www.raspberrypi.com/software/ (visited on 02/14/2023).

[13] *Reliable Kubernetes on a Raspberry Pi Cluster : Storage*. URL: https://github.com/shaposhnikoff/my_medium_articles_starred/blob/master/reliable-kubernetes-on-a-raspberry-pi-cluster-storage.md (visited on 02/14/2023).

[14] *RoboFlow Rat Detection Project*. URL: https://app.roboflow.com/frauas/rat_detection (visited on 02/14/2023).

[15] *Yolov5 GitHub Repository*. URL: https://github.com/ultralytics/yolov5 (visited on 02/14/2023).

[16] *YOLOv5 Object Detection on Windows (Step-By-Step Tutorial)*. URL: https://wandb.ai/onlineinference/YOLO/reports/YOLOv5-Object-Detection-on-Windows-Step-By-Step-Tutorial---VmlldzoxMDQwNzk4 (visited on 02/14/2023).