FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Cloud Computing - Rat Detection Software
Group 5

Amandeep Singh Chhatwal

`amandeep.chhatwal@stud.fra-uas.de`
`Matriculation Number: 1386396`

Rajapreethi Rajendran

`rajapreethi.rajendran@stud.fra-uas.de`
`Matriculation Number: 1396483`

Shounak Ozarkar

`shounak.ozarkar@stud.fra-uas.de`
`Matriculation Number: 1386299`

Mansi Patil

`mansi.patil@stud.fra-uas.de`
`Matriculation Number: 1388967`

Sneha Srinivasa

`sneha.srinivasa@stud.fra-uas.de`
`Matriculation Number: 1400476`

Mrinal Tyagi

`mrinal.tyagi@stud.fra-uas.de`
`Matriculation Number: 1383988`

Faculty 2,
Computer Science & Engineering

February 8, 2023

Contents

1 Introduction

Cloud computing offers a wide range of services like storage, databases, networking and processing power, software, analytics, and intelligence—over the Internet ("the cloud") to offer faster innovation and flexible resources.

Cloud computing can be used to implement a Raspberry Pi-based object detection system. A Raspberry Pi is a low-cost, single-board computer that can be used to run machine learning algorithms and other software applications. The Raspberry Pi can be connected to various sensors and camera to collect data and store it in the cloud.

In a rat detection system, the Raspberry Pi can be connected to camera module to capture image or video. The images can be processed through trained machine learning model to detect and identify rats in the images. The results of the rat detection is then be stored in MinIO server on the cloud for further analysis, reporting and to display the detected image on the UI.

This data combined with image to provide comprehensive picture of the rat population can further be used to train the ML model for more accurate detection and rats behavior in the environment.

The use of cloud computing in a rat detection system provides several advantages, including scalability, accessibility, and cost-effectiveness. In Addition, cloud computing eliminates the need for expensive hardware and storage solutions, making the rat detection system more affordable and accessible.

2 Architecture Diagram

In the Architecture diagram shown below, the edge node runs the machine learning algorithm locally and computes the desired output. This setup is not cluster managed as the industry standard is to use multiple sensor nodes to capture and process data. This is a fundamental concept used in Edge computing. In case of failure of the sensor node, the other sensors nodes running in edge computing topology captures the objects and does the required computation. The local processing of data on the edge node itself delivers faster and efficient results and avoids the latency which would be introduced by sending the data back and forth into the cluster for processing.

The UI application has been build in Flask which renders the results from the database with confidence intervals of the images detected as rats.

The use of CLI is quite efficient to check the health of the cluster in general and to monitor the status of different nodes, pods and services running as a part of the cluster.

Inside the cluster, there is one master node and 3 worker nodes running. The master node manages the services currently active and manages their lifecycle. These include the health of the pods, managing the load balancing services for incoming traffic. The architecture follows the convention of using single container inside one pods until explicitly needed. The real benefit of cloud infrastructure comes from running multiple pods with single containers to achieve the real scalability features. The MinIO runs only in one POD as the Single Node Single Drive mode of deployment is used. For the Ui applications the multiple pods are running to share the load of the incoming traffic.

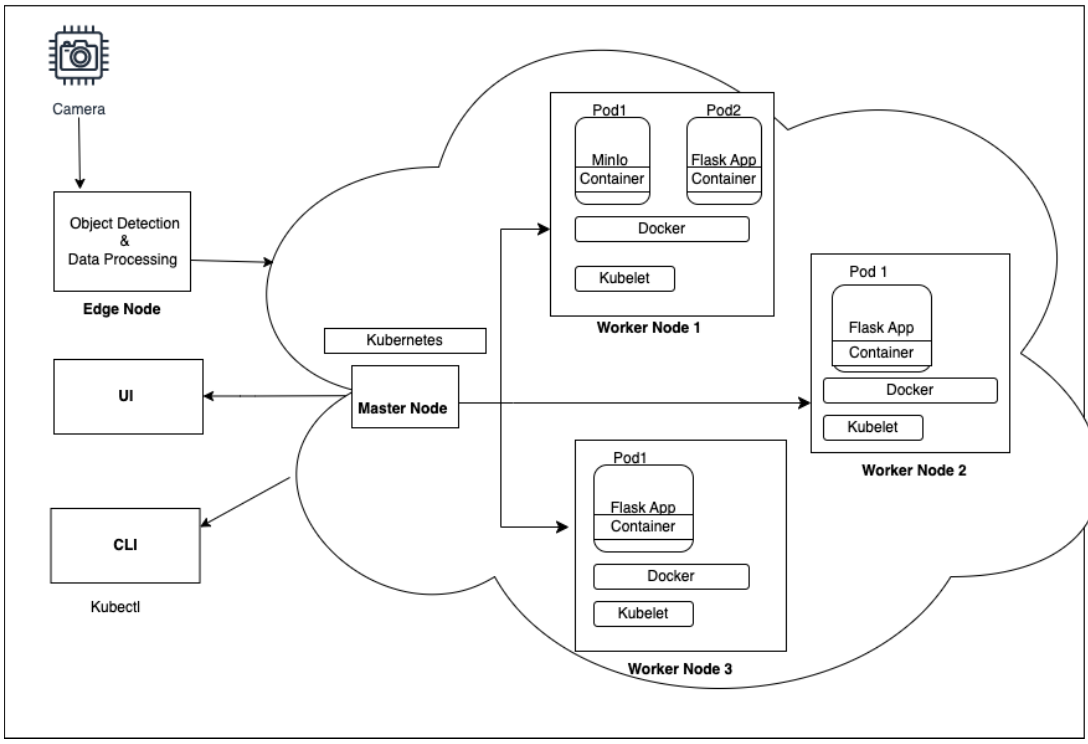


Figure 1: Architecture Diagram

3 Hardware and Software Configuration

3.1 Hardware Configuration in Edge Node

3.1.1 Raspberry Pi 4

The Raspberry Pi 4 is a single-board low-cost, high-performance computer developed by Raspberry Pi Foundation. It has a wide range of applications such as home automation, computer vision, IoT, Artificial Intelligence, etc. Raspberry Pi 4 can run several operating systems such as Raspbian, Debian Ubuntu, Windows, and Linux. It is compatible with a wide range of programming languages such as Python, C, C++, Java, and many others.

Raspberry Pi 4 with Raspberry Pi 4 BULLSEYE 64-bit Debian Os is used as an edge node to perform live rat detection. As Raspberry Pi 4 an edge node, it helps to reduce the amount of data sent to the cloud, reduce latency, save bandwidth, and improve the speed of data processing.

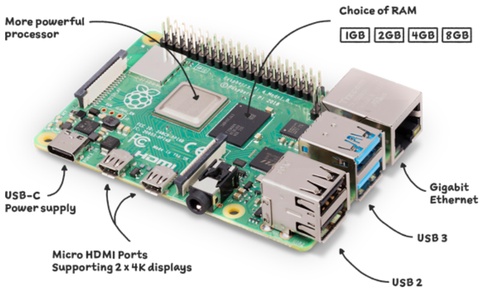


Figure 2: Raspberry Pi 4 [1]

3.1.2 Raspberry Pi Camera Module v2

The Raspberry Pi Camera Module v2 is the official camera module from the Raspberry Pi foundation. It is capable of capturing high-quality video and still images. The Camera Module v2 comes with a Sony IMX219 8-megapixel sensor, which provides a large field of view and high-resolution images.

Camera module v2 is used to capture frames from the live feed for rat detection. It is connected to the Raspberry Pi via a ribbon cable.

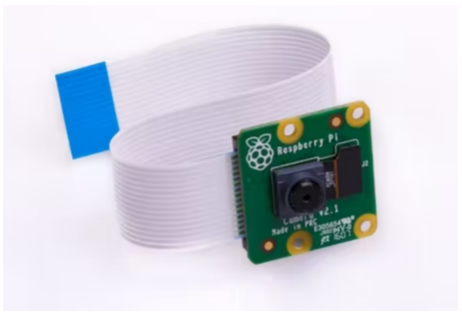


Figure 3: Raspberry Pi Camera Module v2 [2]

3.2 Software Configuration in Edge Node

3.2.1 Python

Python is a high-level, popular interpreted programming language for object detection in computer vision. Raspberry Pi supports Python and it is pre-installed on Raspberry Pi's OS.

The object detection frameworks such as TensorFlow, PyTorch, and OpenCV, provide Python APIs to easily train and deploy object detection models. It has many libraries and packages that are specifically designed for the Raspberry Pi including libraries for accessing the GPIO pins, controlling the camera module and working with sensors.

3.2.2 OpenCV

OpenCV is a popular open-source computer vision library that is well-suited for use on the Raspberry Pi4. It provides a large collection of algorithms and functions for image and video processing, including object detection.

To use OpenCV on the Raspberry Pi 4, the OpenCV library is installed to capture live feeds and to perform object detection. The advantage of using OpenCV on the Raspberry Pi 4 is its performance. The Raspberry Pi 4 has a more powerful CPU and GPU compared to previous Raspberry Pi models, which makes it well-suited for more demanding computer vision tasks.

3.2.3 YOLOv4

YOLOv4 (You Only Look Once version 4) is a state-of-the-art real-time object detection model that is widely used in computer vision applications. Although YOLOv4 requires a powerful GPU. It can be used on the Raspberry Pi 4 for object detection.

It requires an OpenCV library that provides the implementation of YOLOv4. In our project, the YOLOv4 algorithm is used to train and detect rats from the live camera feed.

3.2.4 MinIO Client Library

Minio Client library is a software library for Minio, an object storage server. The Minio Client library provides simple APIs to access Minio and perform various operations such as listing buckets, creating and deleting objects.

3.3 Hardware Configuration in Cluster

3.3.1 Raspberry Pi 3 model

The Raspberry Pi 3 model forms the basis of the cluster hardware. There are total 4 devices which are of type Raspberry Pi 3 which are used in the cluster. One of the device is used as a master and other three devices are used to setup the worker nodes. These models are setup with 32 GB micro sd cards on the appropriate OS will be made available.

3.3.2 TP-LINK Switch

The hardware also consists of the TP-Link Switch for connecting the devices via the Ethernet ports for connectivity. All the Raspberry Pi are connected via Ethernet ports to the splitter.

3.3.3 Fritz-Box

Fritz Box is used to establish the services of the Router for configuring the IP addresses of the devices and providing the DHCP link for the network.

3.4 Installing necessary software components on the cluster

This section covers the software deployment and configurations done to the devices before the setting up of the cluster.

1. Raspberry Pi imager is used install the OS on the SD card. The OS used is headless (lite) version of 64 bit Debian Bullseye without the desktop environment. Using the full version did overload the devices and performance deteriorated, therefore the decision to stick with the headless version is made. The SSH connection details can be added during the OS image setup as this will be later needed to connect the PI over SSH protocol for access
2. Once the Image is setup on the SD card, it needs to be plugged into the PI and once the PI is ready, it is then identified onto the network via the router. After successful login via SSH, the installation is completed by running the update of the packages via update and upgrade commands. It is done in order to ensure the availability of the latest packages for use.
3. After the update, the */boot/cmdline.txt* file needs to be updated. The following text *cgroup_memory=1 cgroup_enable=memory* need to be appended at the end of the file. After the save, the node needs to be restarted.
4. The setting up of the static IP for all the nodes in the cluster is also needed. This is important to ensure that every time the Raspberry PI boots up and connects to the network, it should retain the same IP address. This is relevant for having the stable configurations of the system. The */etc/dhcpcd.conf* file needs to be updated for setting the static IP configurations on each of the available nodes (master as well as worker nodes).

4 Implementation

4.1 Edge Node and Camera Module Set Up for Object detection

To set up Raspberry Pi 4 as an Edge Node with the Pi Camera Module v2, the following steps are followed.

1. **Install and boot the Raspbian Os**: Raspberry Pi 4 BULLSEYE 64-bit Debian Os is installed in the SD card and the SD card is inserted into Raspberry Pi 4. The Raspbian OS should start up and boot into the desktop environment. Raspberry Pi 4 Desktop is accessed using SSH via PuTTY and by VNC Viewer.

2. **Connect the Pi Camera Module v2**: Camera module is connected to the camera port on the Raspberry Pi 4 and the camera interface is enabled in Raspberry Pi Configuration and the raspi-config needs to be updated with *start_x = 1*

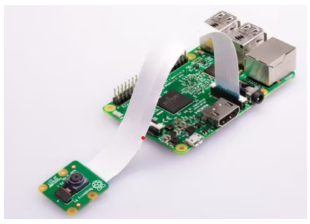


Figure 4: Raspberry Pi Camera Module v2 [2]

3. **Install OpenCV and Python Packages** Python is preinstalled in Raspberry Pi 4 as the Desktop version of Raspberry Pi OS is used. Python dependency such as NumPy and OpenCV 4.7.0 version is installed in order to perform object detection. To upload the detected image to MinIO, The Minio Client library for Python is installed. This will allow interact with the Minio server from the Python code.
   For NumPy installation, the following commands are used.
   (a) Install the pip package manage
   ```
   # sudo apt-get install python3-pip
   ```
   (b) Use pip to install Numpy
   ```
   # pip3 install numpy
   ```
   For OpenCV Installation following commands are executed [5]
   (a) Install minimal prerequisites
   ```
   # sudo apt update  sudo apt install -y cmake g++ wget unzip
   ```
   (b) Download and unpack sources
   ```
   # wget -O opencv.zip https://github.com/opencv/opencv/archive/4.x.zip unzip opencv.zip
   ```
   (c) Create build directory
   ```
   # mkdir -p build  cd build
   ```
   (d) Configure
   ```
   # cmake ../opencv-4.x
   ```
   (e) Build
   ```
   # cmake --build .
   ```
   For MinIO python library installation, the following command is used.
   (a) **Use pip to install Minio Client Library**
   ```
   # pip install minio
   ```

By following steps, Raspberry Pi 4 is set up as an Edge Node and Pi Camera Module v2 is enabled for live video capture. Further Python and OpenCV libraries will be used for object detection and image upload.

4.2 Cluster Set up

4.2.1 Setting the master node using the docker container

The raspberry cluster can be setup in different modes. For this project, the use of docker as the runtime container is preferred as it is widely used as a standard for creating images for running the container in the Pods. The k3s has the default container runtime of *containerd*. This can be changed by specifying the option of docker runtime while setting the k3s on each of the nodes. The docker runtime must be made available before its use via one of Rancher scripts using the following command

```
# curl https://releases.rancher.com/install-docker/20.10.sh | sh
```

Once the docker is available, then the setup of k3s on the nodes can be done. The use of the following command install the k3s on the master node

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

Once the k3s service is up and running, the next step would be to extract the token from the master node to be then used while setting the worker nodes. The token is available at the location */var/lib/rancher/k3s/server/node-token*

4.2.2 Setting the worker nodes using the docker container

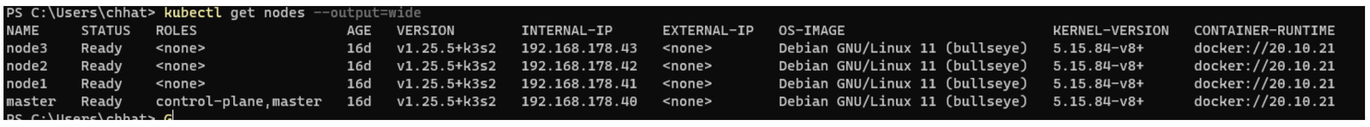
The use of the following command sets up the worker node k3s service. The docker runtime assumed to be available on the worker node and in case it is not available, please refer the section above to download the docker runtime.

```
# curl -sfL https://get.k3s.io | K3S_URL=https://<ip-address>:6443 K3S_TOKEN=<node-token>
sh -s - --docker
```

The *ip-address* needs to be replaced with the IP of the Raspberry Pi where the master node has been setup. The port number 6443 is the default port where the service is running for k3s. The *node-token* needs to be replaced by the node-token captured from master node at the location mentioned in the above section. This same process needs to be repeated for all the worker nodes that need to be setup as part of the cluster. Once all the services are up and running, the status of the cluster can be requested by the following command.

```
# kubectl get nodes --output=wide
```

The command will render similar results as shown below



```
PS C:\Users\chhat> kubectl get nodes --output=wide
NAME     STATUS   ROLES                  AGE   VERSION        INTERNAL-IP      EXTERNAL-IP   OS-IMAGE                         KERNEL-VERSION   CONTAINER-RUNTIME
node3    Ready    <none>                 16d   v1.25.5+k3s2   192.168.178.43   <none>        Debian GNU/Linux 11 (bullseye)   5.15.84-v8+      docker://20.10.21
node2    Ready    <none>                 16d   v1.25.5+k3s2   192.168.178.42   <none>        Debian GNU/Linux 11 (bullseye)   5.15.84-v8+      docker://20.10.21
node1    Ready    <none>                 16d   v1.25.5+k3s2   192.168.178.41   <none>        Debian GNU/Linux 11 (bullseye)   5.15.84-v8+      docker://20.10.21
master   Ready    control-plane,master   16d   v1.25.5+k3s2   192.168.178.40   <none>        Debian GNU/Linux 11 (bullseye)   5.15.84-v8+      docker://20.10.21
PS C:\Users\chhat>
```

Figure 5: Cluster status

4.2.3 Setting up of the kubectl utility

It is efficient to use the kubectl utility without logging physically into the Raspberry Pi. For this, the contents of the following file */etc/rancher/k3s/k3s.yaml* from the master node needs to be copied to the local kubectl config file. The Server https address needs to be updated from *https://127.0.0.1:6443* to the actual IP of the raspberry pi running the master node. Important commands used during the project timespan can be found at https://github.com/AmandeepChhatwal/FindTheRats/blob/main/Read_Me_file_for_all_cluster_setup.docx
The cluster setup can look something like the following, once everything is up and running.

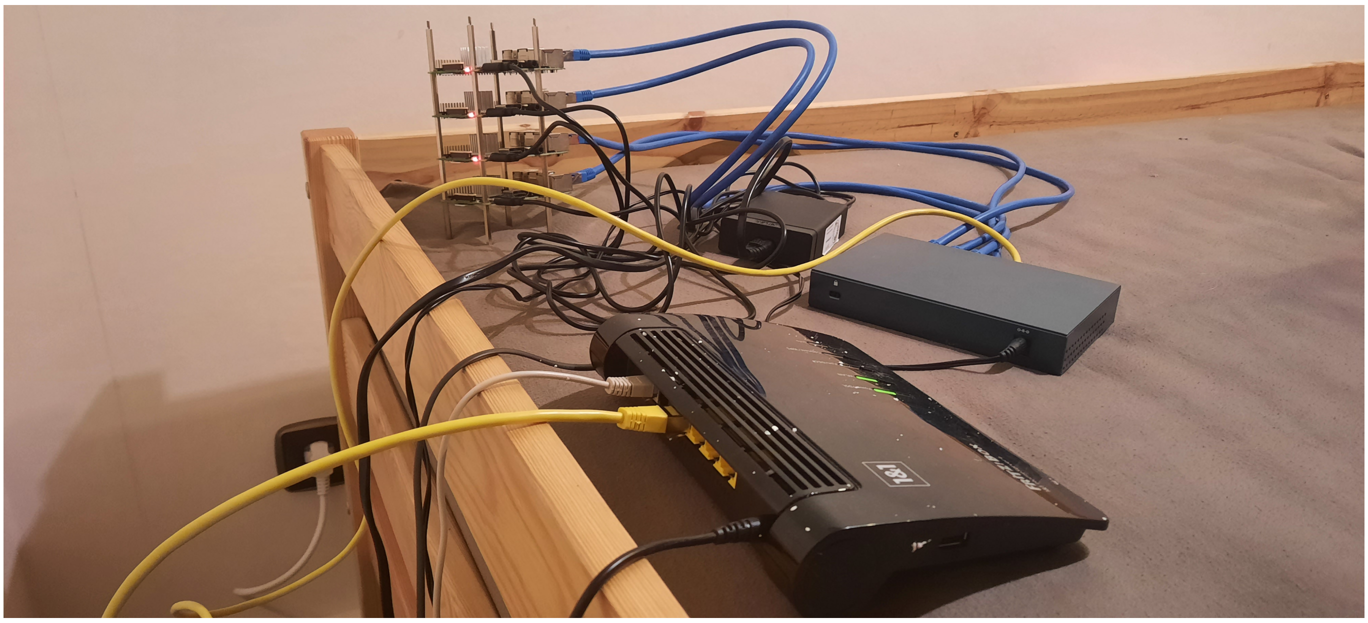


Figure 6: Cluster setup

4.3 Deployment of MinIO in Cluster

The database of MinIO is chosen as it is specifically designed to handle the storage of objects especially images which is the case in this image detection software. There are different factors which have been considered before the use of MinIO. The scalability, resilience and the availablity of RESTful API tailor made for use in Kubernetes cluster are some of those features of MinIO that are really useful. The deployment of MinIO is done via the deployment artifact and a service is made available to expose the access to the command line interface. There are mutiple modes in which MinIO can be setup in the cloud network [8]. In the project developed, the use of Single Node Single Drive mode has been used. The Multi node Multi Drive features require more resourceful hardware for ensuring the synchronization between multiple instance of the database running on the cluster. This is tried out but because of the challenges faced and keeping in mind the time constraints of the project, the single drive single node mode is implemented. The entire information is found in the YAML file available at the location on the GitHub at https://github.com/AmandeepChhatwal/FindTheRats/tree/main/minio

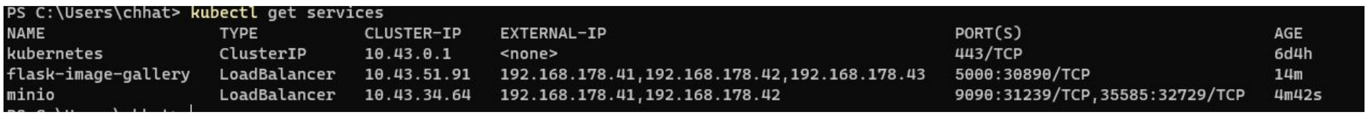
4.3.1 Setting Minio over Kubernetes Service

The availability of the Minio exist over a Kubernetes service which is useful for testing and integration purposes. The use of Service helps to access the Command line UI from outside the cluster network. In this project, the use of RESTful API from MinIO helps to connect the Edge Node for uploading the images to the MinIO database directly.

4.3.2 Enabling minio with persistent volumes

The use of Persistant volumes is advocated in order to ensure the availability of data in case of scenarios where the POD running the minio server crashes because of any reasons. This ensures the preventation of data loss in cloud based infrastructures. Once the service is up and running, it could be accessed with the following command

```
# kubectl get services
```



```
PS C:\Users\chhat> kubectl get services
NAME                  TYPE           CLUSTER-IP     EXTERNAL-IP                                    PORT(S)                          AGE
kubernetes            ClusterIP      10.43.0.1      <none>                                         443/TCP                          6d4h
flask-image-gallery   LoadBalancer   10.43.51.91    192.168.178.41,192.168.178.42,192.168.178.43   5000:30890/TCP                   14m
minio                 LoadBalancer   10.43.34.64    192.168.178.41,192.168.178.42                  9090:31239/TCP,35585:32729/TCP   4m42s
```

Figure 7: Service status

4.4 Model Training

For object detection we used YOLOv4 [5]. YOLO is short for You Only Look Once. It is a real-time object recognition system that can recognize multiple objects in a single frame. YOLO recognizes objects more precisely and faster than other recognition systems.

1. The utmost important thing for object detection is having a good quality labeled images for training data. The label data set can be found on https://universe.roboflow.com/ which may suffice for the needs or label data can be set using one the tools available. Recommendation would be https://github.com/developer0hye/Yolo_Label
2. After labeling you should have a text file containing the coordinates of the box for the labels. Based on the version of YOLO you are using you'll have different label files
3. Split the data into train and test folder (code snippet available at the link https://stackoverflow.com/questions/66579311/yolov4-custom-dataset-train-test-split can be used to split this into 2 folders)

4. After we have our data set ready we move on to training the model. Easiest Option is to create a clone of Google Collab notebook for training the model as we have a dedicated GPU available for certain period of time. We can do this on a system with a GPU as well. Around 3000 rat images and their labels were used to train the model.
5. Update the config file present named as coco.yaml used during training to refer the training validation data and also to check the classes for which we are training the model. Since we trained the model only for 1 class i.e. rat class, remove the other classes from this configuration file.
6. Inside the google collab, we follow the steps mention and make certain changes for dataset and the classes that we are detecting In the collab
   (a) Mount google drive (this make data in the drive available inside the collab)

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

   (b) Upload your train and test data folders in drive
   (c) Clone darknet git repository

   ```
   ! git clone https://github.com/AlexeyAB/darknet.git
   ```

   (d) Compile darknet.exe

   ```
   !make clean
   !make
   !chmod +x ./darknet
   ```

   (e) Make appropriate changes in yolov4 config file (Details mentioned below) then Download the pre-trained yolov4 weights
   (f) Start the training of custom detector

   ```
   !./darknet detector train data/yolov4.data
       cfg/yolov4_custom_train.cfg
       backup/yolov4_custom_train_last.weights -dont_show -map
   ```

   (g) Test the model on new image

   ```
   img_path = "./example/d37ad1e042b16886.jpg"
   !./darknet detect cfg/yolov4_custom_train.cfg
       backup/yolov4_custom_train_best.weights {img_path} -dont-show
   ```

Detailed steps of the above process are mentioned in the jyputer notebook available at https://github.com/AmandeepChhatwal/FindTheRats/blob/main/ratid_lab.ipynb

After training is complete, download weights, configuration, and classes of the trained model as we'll need to use it for object detection

4.4.1 Idea behind tuning the config parameters:

Subdivisions are the numbers of mini-batches the complete batch is spit into. To give an example a batch=52 denotes that 52 images will be loaded for one iteration. Henceforth, the subdivision=4 denotes that 52/4=13 images per mini-batch and these 13 images will be sent for processing. This will be repeated till the whole batch is completed. The size of subdivision also depends on GPU, as for low memory, subdivisions are set a higher value and visa versa. This brings us to the initial changes in the yolov4-custom.cfg file. we have trained the dataset on the following parameters. Our explanation of the parameter is written as comment infron of it

```
classes= 1, # number of classes of objects which can be detected
max_batches=3000 # the training will be processed for this number of
   iterations (batches)
batch=64 # number of samples (images) which will be precossed in one batch
subdivisions=16 # The batch is subdivided in this many "blocks". The images
   of a block are ran in parallel on the gpu
width=416 # network size (width), so every image will be resized to the
   network size during Training and Detection
height=416 # network size (height), so every image will be resized to the
   network size during Training and Detection
channels=3 # network size (channels), so every image will be converted to
   this number of channels during Training and Detection

momentum=0.949 # accumulation of movement, how much the history affects the
   further change of weights (optimizer)
decay=0.0005 # a term to diminish the weights to avoid having large values.
   For stability reasons I guess.
learning_rate=0.001
steps= (640,720) # this is a checkpoints (number of iterations) at which
   scales will be applied.
scales=(0.1,0.1) # scales is a coefficients at which learning_rate will be
   multiplied at this checkpoints. Determines how the learning_rate will be
   changed during increasing number of iterations during training.
```

4.5 Rat Detection in Edge Node

The “Object detection using deep learning with OpenCV and Python” [4] is taken as a reference and the rat detection is developed. The rat detection in the edge node is implemented by multithreading using Python and OpenCV. The below architecture explains the rat detection at edge node.
Thread 1 is a capture thread that captures the live feed and processes one frame per second and adds it to the queue. Thread 2 is a detection thread that fetches a frame from the queue and passes it to the YOLOv4 detection model. In the detection model, the model’s weights, configuration, and classes are given and the network is loaded into

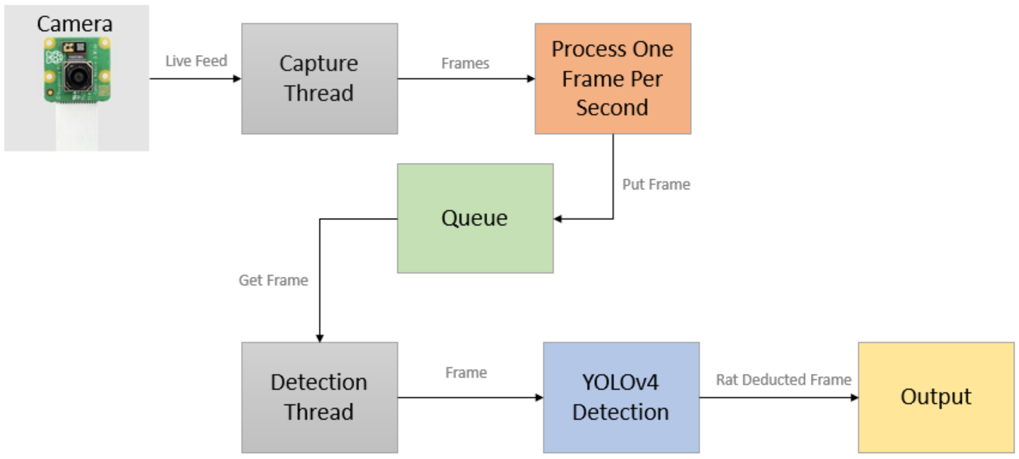


Figure 8: Rat Detection at Edge Node

the OpenCV DNN(Deep Neural Network) module.

In the OpenCV DNN module, the output layers for object detection tasks are typically calculated by forward propagating an input image through a pre-trained deep neural network model. The output layers of the network produce a set of predictions that describe the locations and class probabilities of objects in the image. The model then returns the bounding boxes around the detected rat along with the class label rat and confidence scores. The below figure is the detected rat at the edge node which is shown with the class label rat and the model predicts it with a 72% confidence level.

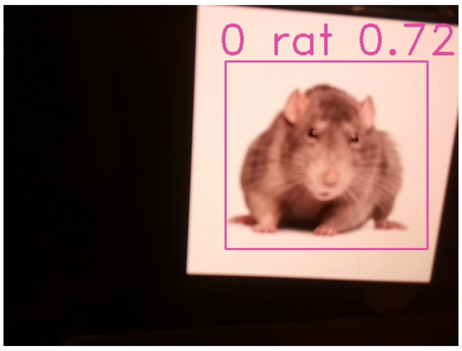


Figure 9: Detected Rat

The codebase for the object detection can be found at the following path
https://github.com/AmandeepChhatwal/FindTheRats/tree/main/ObjectDetection

4.6 Detected Frame upload to MinIo

The detected image from the object detection script is uploaded to the MinIO server which is hosted in K3s Cluster. The below figure shows the end-to-end process of detecting the rat from the camera and uploading the image to the MinIO server in the K3s cluster.

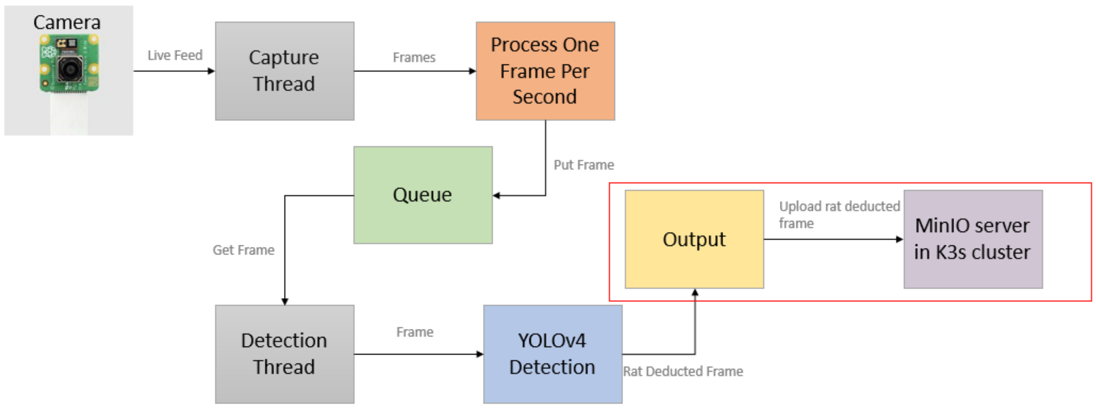
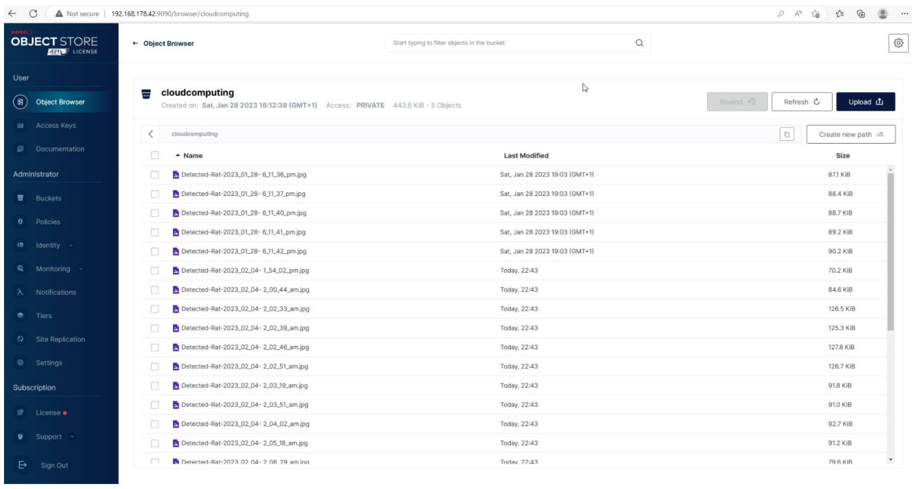


Figure 10: Frame Upload to MinIO

To upload the rat deducted frame, MinIO client object is created that represents the MinIO server hosted in the K3s cluster. The MinIO server endpoint, access key, secret key, and security level are set to the MinIO client object [7]. To store the image in the MinIO server, buckets needs to be created. Using the MinIO client the bucket can be created and using the put_object method the images are uploaded. The below picture shows the rat-detected frame from edge node which got uploaded to MinIO server hosted in k3s cluster



cloudcomputing
Created on: Sat, Jan 28 2023 16:12:39 (GMT+1) Access: PRIVATE 443.6 KiB - 5 Objects

cloudcomputing

Name	Last Modified	Size
Detected-Rat-2023_01_28- 6_11_36_pm.jpg	Sat, Jan 28 2023 19:03 (GMT+1)	87.1 KiB
Detected-Rat-2023_01_28- 6_11_37_pm.jpg	Sat, Jan 28 2023 19:03 (GMT+1)	88.4 KiB
Detected-Rat-2023_01_28- 6_11_40_pm.jpg	Sat, Jan 28 2023 19:03 (GMT+1)	88.7 KiB
Detected-Rat-2023_01_28- 6_11_41_pm.jpg	Sat, Jan 28 2023 19:03 (GMT+1)	89.2 KiB
Detected-Rat-2023_01_28- 6_11_42_pm.jpg	Sat, Jan 28 2023 19:03 (GMT+1)	90.2 KiB
Detected-Rat-2023_02_04- 1_54_02_pm.jpg	Today, 22:43	70.2 KiB
Detected-Rat-2023_02_04- 2_00_44_am.jpg	Today, 22:43	84.6 KiB
Detected-Rat-2023_02_04- 2_02_33_am.jpg	Today, 22:43	126.5 KiB
Detected-Rat-2023_02_04- 2_02_39_am.jpg	Today, 22:43	125.3 KiB
Detected-Rat-2023_02_04- 2_02_46_am.jpg	Today, 22:43	127.8 KiB
Detected-Rat-2023_02_04- 2_02_51_am.jpg	Today, 22:43	126.7 KiB
Detected-Rat-2023_02_04- 2_03_19_am.jpg	Today, 22:43	91.8 KiB
Detected-Rat-2023_02_04- 2_03_51_am.jpg	Today, 22:43	91.0 KiB
Detected-Rat-2023_02_04- 2_04_02_am.jpg	Today, 22:43	92.7 KiB
Detected-Rat-2023_02_04- 2_05_18_am.jpg	Today, 22:43	91.2 KiB
Detected-Rat-2023_02_04- 2_08_29_am.jpg	Today, 22:43	79.6 KiB

Figure 11: Frame in MinIO Server

4.7 Web Application Implementation

To display the images of the rat captured through the camera which are then subsequently stored in MinIO Database we have created a web application.

A simple python flask application can be to construct an User Interface. Flask is a small and lightweight Python web framework that provides useful tools and features that make creating web applications in Python easier.

4.7.1 Installation of Required Software

1. **Install Python**
   https://www.python.org/downloads/release/python-3111/
2. **Install a text editor**
   https://code.visualstudio.com/download
3. **Start a new project with VirtualEnv**
   Create the Python Virtual Environment and then activate the environment.
   ```
   # python3 -m venv venv
   # source venv/bin/activate
   ```
4. **Install Requirements**
   Install the Flask Minio module under the virtual environment.Mention the modules inside requirments.txt
   ```
   # pip install -r requirements.txt
   ```
   Create a flask Application and write the code in .py file and run with the python command.
   ```
   # python3 app.py
   ```

The images displayed into the web app are read from the folder rat-images and the images already displayed on the web app are moved to the archive folder. The codebase for the

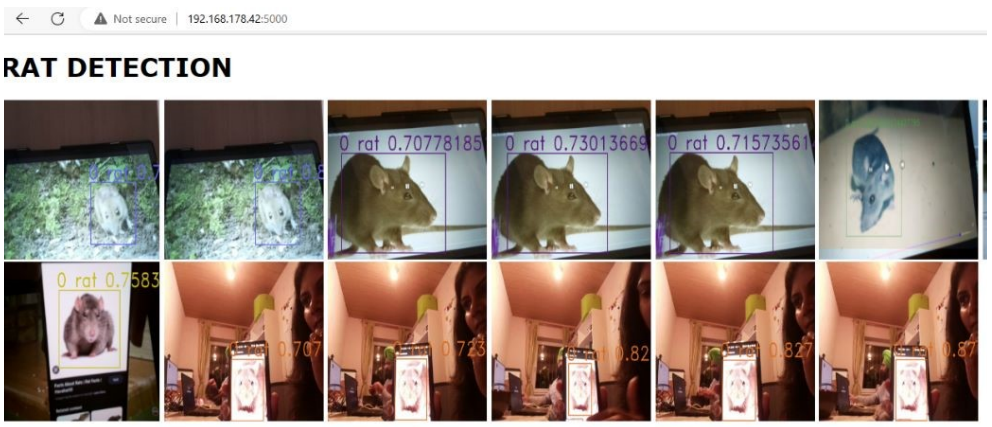


Figure 12: User Interface

web application can be found at the following path
https://github.com/AmandeepChhatwal/FindTheRats/tree/main/ImageGallery/Flask-Image-Gallery-

4.7.2 Containerization and Deployment of the web app

Once the web application is running successfully. The next step is to create an Image of the web application and run the application in Docker.

The first step to containerizing the application is to create a docker file.

The dockerfile can be found at the below github path:
https://github.com/AmandeepChhatwal/FindTheRats/blob/main/ImageGallery/Flask-Image-Gallery
DockerFile

This file is a set of instructions Docker will use to build the image.The instruction are as follows

1. Get the official Python Base Image for version 3.7 from Docker Hub.
2. In the image, create a directory named app.
3. Set the working directory to that new app directory.
4. Copy the local directory's contents to that new folder into the image.
5. Run the pip installer (just like we did earlier) to pull the requirements into the image.
6. Inform Docker the container listens on port 5000.
7. Configure the starting command to use when the container starts. [6]

On the command line or shell, to build the image use the following command:

```
docker buildx build --platform linux/arm64 -t
    <youreponame>/rat-detection:arm64 --push
```

The application is now containerized.Run the following command to have Docker run the application in a container and map it to port 5000:

```
# docker run -p 5000:5000 ratui
```

Now navigate to http://localhost:5000 to see the images of the detected rats

To run the application in Kubernetes, First verify if kubectl is configured, if not, install and configure it.

```
# kubectl version
# kubectl config use-context docker-for-desktop
```

iCreate a file named deployment.yaml and add the following contents to it and then save

The github link for the deployment file is as below :
https://github.com/AmandeepChhatwal/FindTheRats/blob/main/ImageGallery/Flask-Image-Gallery
deployment.yaml

This YAML file is the instructions to Kubernetes for what is required for running, For example,load-balanced service exposing port 6000 and four instances of the container running.

Use kubectl to send the YAML file to Kubernetes by running the following command:

```
# kubectl apply -f deployment.yaml
```

The latest rat Images can be viewed on http://localhost:5000

5 Challenges

There are several challenges and learning while implementing rat detection in Raspberry Pi with edge and cloud infrastructure.

1. In the Edge node, the configuration issues arise when OpenCV tried accessing the camera's live feed. The error was not clear which required the complete reinstallation of OpenCV to resolve the issue.
2. The YOLOv4 object detection model in Raspberry pi 4 results in performance issues due to its limited processing power and it impacted the performance largely. With the use of the YOLOv4 tiny model, the performance is improved but the accuracy is drastically reduced. So the YOLOv4 is used back in object detection but the efficiency of the model is achieved by multithreading.
3. The sharing of the infrastructure resulted in disarray of the configurations because of static IP that were already setup. Therefore, there were multiple runs where the infrastructure had to be setup from the scratch. There is a risk of wear and tear whenever the infrastructure exchanged hands resulted in non-usability of devices or its components.
4. The effective use of the infrastructure also had a learning curve as the given Raspberry Pi are resource constrained devices
5. The use of MinIO in Multi Node Multi Drive mode is not possible because of needed resource capacity for synchronization processes.
6. The use of DHCP server for setting the infrastructure is also a time consuming process. This is eventually solved by using a Fritz-Box in the end.

6 Summary

The implementation of object detection with a K3s cluster cloud infrastructure in Raspberry Pi has several advantages over traditional on-premise solutions. The use of cloud infrastructure provides the ability to scale, improve performance, and enables faster object detection, and it's easily accessible. Additionally, the use of the k3s cluster provides an efficient way to deploy and manage containerized object detection applications and it can easily be integrated with cloud services such as storage and databases. However, there are challenges to consider when implementing object detection with cloud infrastructure such as containerization, expertise in cloud infrastructure, and object detection, and the need to carefully consider data privacy and security issues and regulatory compliance. Overall, the combination of technologies provides a flexible and powerful solution for object detection that can be applied to a wide range of use cases.

References

[1] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[2] https://www.raspberrypi.com/products/camera-module-v2/

[3] https://github.com/AlexeyAB/darknet/

[4] https://github.com/arunponnusamy/object-detection-opencv

[5] https://docs.opencv.org/4.x/d7/d9f/tutorial_linux_install.html

[6] https://towardsdatascience.com/how-to-deploy-a-flask-api-in-kubernetes-and-connect-it-with-oth

[7] https://medium.com/featurepreneur/upload-files-in-minio-using-python-4f987f902076

[8] https://min.io/docs/minio/linux/operations/installation.html