FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Cloud Computing WS22/23

Project Report

Automatic Rat Detection using Edge Computing

Supervisor: Prof. Dr. Christian Baun

Submitted By: Anish Pokhrel (1394715),

Ashlesh Mithur (1386367),

Deepak Kumar (1400489),

Nidhi Nayak (1404524),

Shobhit Tiwari (1387366),

Pushpita Sarkar (1384152),

Arpan Kumar (1378650)

Submission Date: 8th February 2023

Contents

Team Members and Contributions

Task	**Contributors**
Initial Hardware Setup, Testing	Ashlesh Mithur, Arpan Kumar
K3S cluster, Sensor Node setup	Ashlesh Mithur
ML Model, Training, Sensor Node Deployment	Deepak Kumar
API development & DB setup	Anish Pokhrel
User Interface development	Shobhit Tiwari, Nidhi Nayak
Notification and Alerts	Anish Pokhrel, Pushpita Sarkar
Sensor Node and Cluster Integration	Deepak Kumar, Ashlesh Mithur
Project Integration	Anish Pokhrel, Ashlesh Mithur, Deepak Kumar
Documentation	Anish Pokhrel, Ashlesh Mithur, Deepak Kumar, Shobhit Tiwari

Repository: https://github.com/dpk0811/Rat-Detection

Scrum Board: https://trello.com/b/9EJAa3ZV/cloud-computing-project

1. Introduction

For the cloud computing semester project WS22/23, our goal was to develop an edge computing solution to detect rats at the sensor node and store the results in cloud.

In cloud computing, data is sent to the cloud for processing and storage. Edge computing on the other hand, processes data at the edge node and then sends only limited amount of data to the cloud for storage. Edge computing has several benefits over traditional cloud computing approaches. Edge computing offers better latency compared to cloud computing as the data is processed near the source i.e., at the edge node. Edge computing solutions also require lesser bandwidth as the data is processed at the edge node. Edge computing solutions are scalable, reliable, cost effective and offer better privacy and security.

2. Architecture

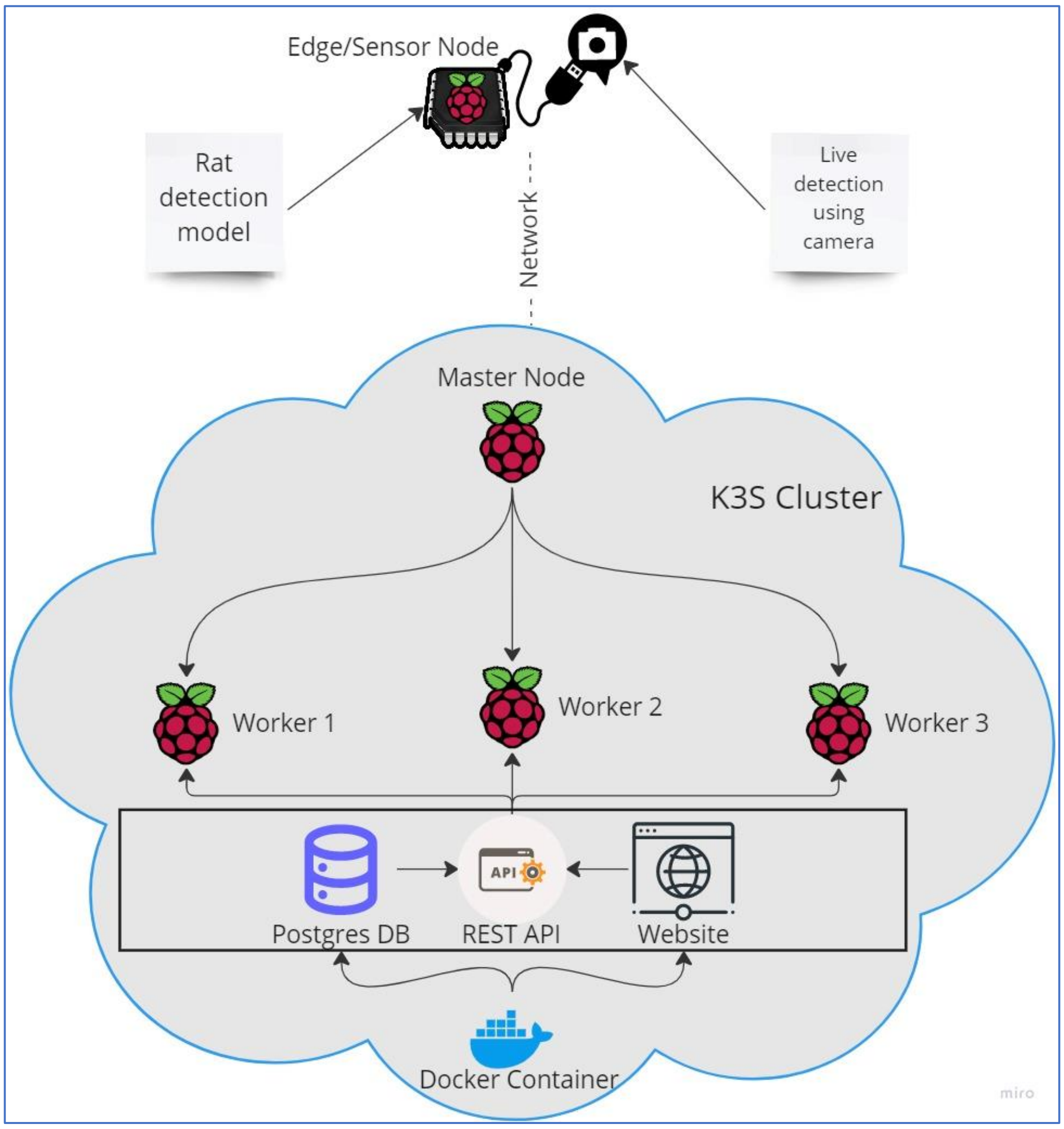


Figure 1: System Architecture

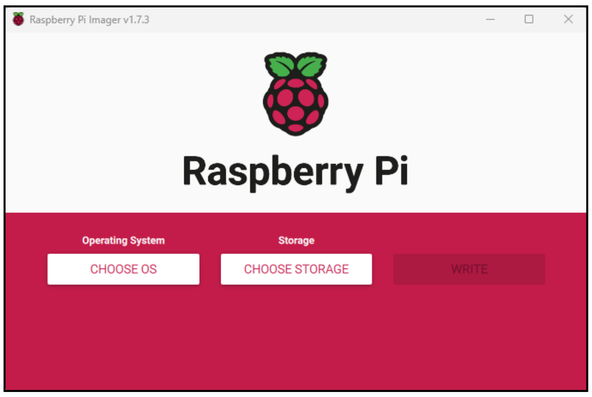
Our rat detection project is built as an edge computing solution. We used a Raspberry Pi 4 single board computer (SBC) as the edge/sensor node on which our machine learning model trained to detect rats was deployed. The SBC is also equipped with a Raspberry Pi 2 camera module. We also have a K3S cluster built using 4 Raspberry Pi 3 SBC's where 1 SBC behaves as the master node and the rest 3 behave as the worker nodes. Live detection is ran using the camera module at the edge node. Whenever a rat is detected, the frame is captured and is sent to the K3S cluster. On the K3S cluster, we have setup a postgres database which is used to store data relevant to the detection such as confidence level, timestamp of the capture and the image frame itself. We have also setup a python flask web application on the K3S cluster, which is used to view the detected images. To send the image from the edge node to the K3S cluster, we have used REST API's. The detected images along with the relevant data are sent to the REST API. Upon receiving data, the REST API pushes the data to the postgres database and triggers a notification. For notification, we are using Slack. With every detection, a slack notification is sent which informs the user about the number of rats detected along with the highest confidence level of detection.

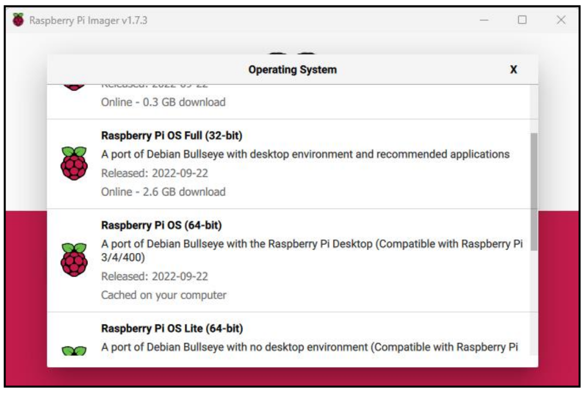
3. Sensor Node

Sensor node is a Raspberry Pi 4 single board computer (SBC) with an attached Raspberry Pi camera module 2. In our project, the sensor node is used as an edge device to detect rats, build relevant data and send it over to the REST API running on K3S cluster.

3.1.Sensor Node Setup

To setup the sensor node, we installed Raspberry Pi OS 64 bit using the Raspberry Pi Imager tool.




Raspberry Pi Imager v1.7.3

Operating System

Online - 0.3 GB download

Raspberry Pi OS Full (32-bit)
A port of Debian Bullseye with desktop environment and recommended applications
Released: 2022-09-22
Online - 2.6 GB download

Raspberry Pi OS (64-bit)
A port of Debian Bullseye with the Raspberry Pi Desktop (Compatible with Raspberry Pi 3/4/400)
Released: 2022-09-22
Cached on your computer

Raspberry Pi OS Lite (64-bit)
A port of Debian Bullseye with no desktop environment (Compatible with Raspberry Pi

The OS can be selected from the ***Choose OS*** option on the tool. The storage i.e., the SD card was set using the ***Choose Storage*** option of the tool. Once both these are set, we can write the OS on the SD card using the ***Write*** Option of the tool.

The SD card was then inserted back into the Raspberry Pi 4 SBC.

4. Machine Learning Model

For automatic rat detection on the sensor, we trained a machine learning model using YOLOv7 framework.

YOLOv7 (You Only Look Once) is a real-time object detection framework which can detect multiple objects. By default, YOLOv7 is trained to detect over 70 classes which include most of the common objects encountered in day-to-day life. For our scenario i.e., detection of rats, the YOLOv7 model had to be retrained.

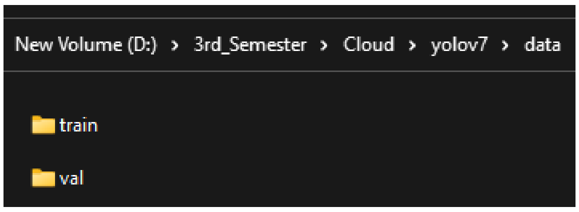
Google Colab was used as it offers free computing resources to train machine learning models. We started with scraping rat images for training and validating the model. Around 4500 rat images and their labels were used to train the model.

Training a YOLOv7 model is very straight forward. We started with cloning the YOLOv7 repository from GitHub using the below command,

git clone https://github.com/WongKinYiu/yolov7.git

The repository comes with python scripts to train, test and validate the model and also to perform detections after the model has been built.

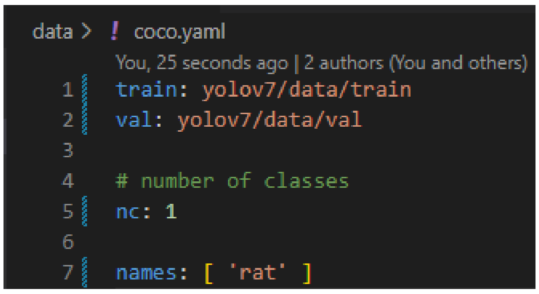
The entire dataset of 4500+ rat images and their labels were split into training and validation datasets in a 70:30% ratio. After this they were placed into their respective folders in the YOLOv7 cloned structure i.e., ***data\train*** & ***data\val***.


New Volume (D:) › 3rd_Semester › Cloud › yolov7 › data

train

val

In the next step, we updated the config file present at ***data\coco.yaml*** used during training to find 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, we removed other classes from this configuration file.


data > ! coco.yaml

You, 25 seconds ago | 2 authors (You and others)

```yaml
train: yolov7/data/train
val: yolov7/data/val

# number of classes
nc: 1

names: [ 'rat' ]
```

We then uploaded the YOLOv7 cloned structure to Google Drive for easier accessibility with Google Colab.

To begin training the model in Google Colab, we first need to link it with Google Drive. Run the below lines in Google Colab,

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

Once Google drive is mounted successfully, we need to install all the python libraries needed by YOLOv7 during model training. YOLOv7 already provided a ***requirements.txt*** containing the list of libraries within in. We can use this file to install everything at once with the below command,

```
!pip install -r drive/MyDrive/yolov7/requirements.txt
```

If we have access to GPU, we can train the model even faster and to do this, YOLOv7 needs few additional libraries which are mentioned in the ***requirements_gpu.txt***. To install these libraries, run the below command,

```
!pip install -r drive/MyDrive/yolov7/requirements_gpu.txt
```

Now that all the libraries are installed in Google Colab, we begin training our model. The below command can be used to start model training,

```
!python train.py --workers 1 --device 0 --batch-size 16 --epochs 100 --img 640 640 --
hyp data/hyp.scratch.custom.yaml --name yolov7-custom --weights yolov7.pt
```

The above command takes a lot of arguments which sets up various parameters for training the model.

For ex.:

--img: image size for which model is trained,

--epochs: number of training epochs,

--weights: pre-trained YOLOv7 weights,

--batch-size: batch size used during training, etc.

Free version of Google Colab allocates computing resources for a limited duration every day. In case the model training is interrupted due to this reason, the training session can be resumed from where it stopped using the command,

```
!python train.py --resume --workers 1 --device 0 --batch-size 16 --epochs 100 --img 640 640 --
hyp data/hyp.scratch.custom.yaml --name yolov7-custom --weights yolov7.pt
```

The model training can also be done without using GPU. To do so, the below command can be used,

```
!python train.py --workers 1 --device cpu --batch-size 16 --epochs 100 --img 640 640 --
hyp data/hyp.scratch.custom.yaml --name yolov7-custom --weights yolov7.pt
```

Once the training is completed, we would get the weights file that can be used to perform detections. During the entire training process, several weight files are created in the folder ***runs\train\yolov7-custom\weights***. For detection, we used weight file named ***best.pt***.

Using the below command, we can perform detections on images,

```
!python detect.py --weights best.pt --conf 0.3 --img-size 640 --source test_cat.jpg --no-trace
```

To perform detection on video from Google Colab, we can use the below command,

```
!python detect.py --weights best.pt --conf 0.3 --img-size 640 --source test_video.mkv --no-trace
```

To view statistics related to model training, we can find confusion matrix, p curve, r curve, pr curve etc., in the folder ***runs\train\yolov7-custom\***.

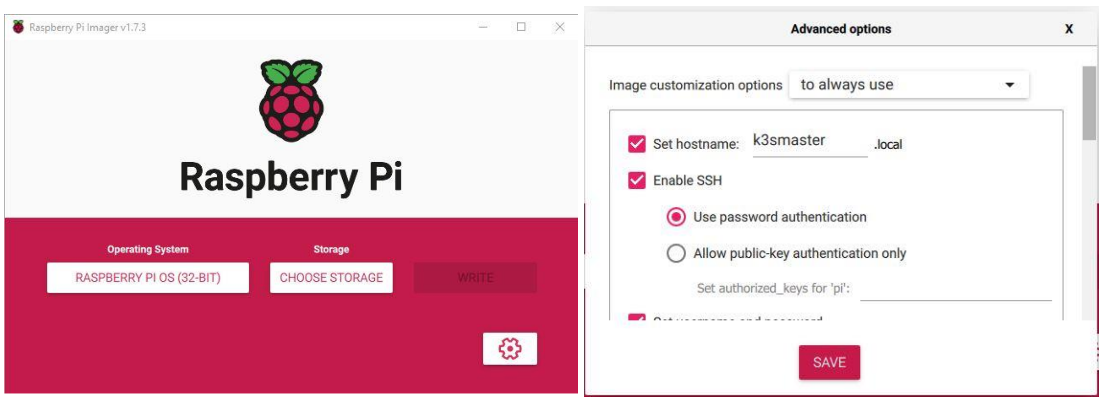
5. Setting up K3S Cluster using Raspberry Pi 3

As discussed in the architecture we had used 4 different Raspberry Pi 3 SBC to setup a light weight Kubernetes cluster or K3S cluster. The cluster was created with 1 master and 3 worker nodes.

Below is initial configuration required for setting up the cluster (windows machine was used during this setup) and followed the article [1].

5.1. Setting up all Raspberry Pi 3

- All the Raspberry Pi 3 was equipped with 32GB SD cards, we manually flashed 32-bit Raspberry Pi OS with help of Raspberry Pi Imager v1.7.3.
- Download Raspberry Pi Imager for your computer and insert an empty SD card to your PC.
- In the Pi Imager application, we chose 32-bit Raspberry Pi OS (Debian Bullseye) and configured the hostname, enabled SSH and also set password for authentication in the advanced options as shown below image.



Operating System: RASPBERRY PI OS (32-BIT)

Image customization options: to always use

[✓] Set hostname: k3smaster .local

[✓] Enable SSH

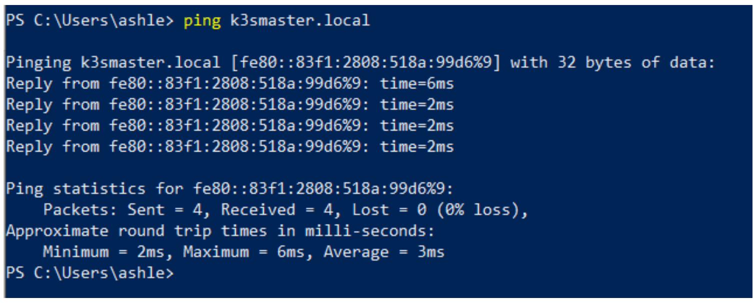
[● Use password authentication]

○ Allow public-key authentication only

Set authorized_keys for 'pi':

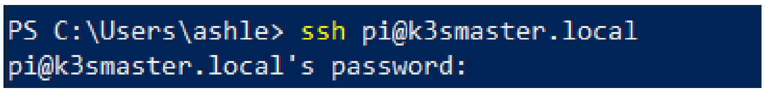
- Once values are configured and saved, select storage option as the SD card and click on 'write' button that fill flash the SD card with chosen OS.
- We repeated this process for all 4 SD cards and named our hosts as k3smaster, ksworker1, ksworker2, ksworker3 respectively.
- Insert all the SD cards back to the Raspberry Pi and power up and connect them to your network via LAN switch.

- We did a ping check to confirm if everything is working as shown in the image below.

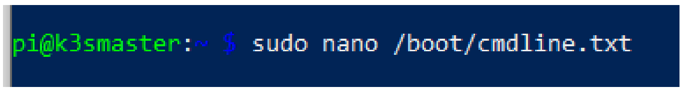


```
PS C:\Users\ashle> ping k3smaster.local

Pinging k3smaster.local [fe80::83f1:2808:518a:99d6%9] with 32 bytes of data:
Reply from fe80::83f1:2808:518a:99d6%9: time=6ms
Reply from fe80::83f1:2808:518a:99d6%9: time=2ms
Reply from fe80::83f1:2808:518a:99d6%9: time=2ms
Reply from fe80::83f1:2808:518a:99d6%9: time=2ms

Ping statistics for fe80::83f1:2808:518a:99d6%9:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 2ms, Maximum = 6ms, Average = 3ms
PS C:\Users\ashle>
```

- Another important step we followed is to enable the **cgroup** memory. SSH into all the Raspberry Pi 3 and update the **cmdline.txt** file.
- We used "*sudo nano /boot/cmdline.txt*" command in all the hosts and added "*cgroup_memory=1 cgroup_enable=memory*" on the end of the first line and save the file.



```
PS C:\Users\ashle> ssh pi@k3smaster.local
pi@k3smaster.local's password:
```

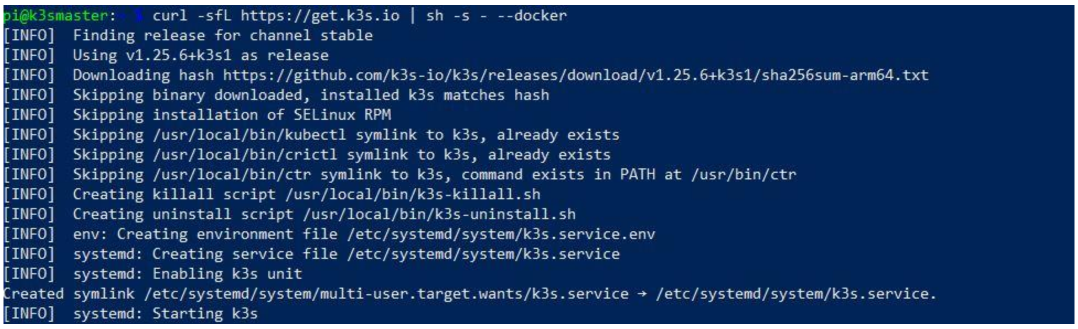


```
pi@k3smaster:~ $ sudo nano /boot/cmdline.txt
```

- Now reboot using "*sudo reboot*" and start again.

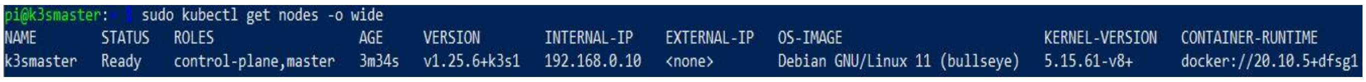
5.2. Setting up k3s cluster

- SSH into master node, k3smaster
- Use the command:
  curl -sfL https://get.k3s.io | sh -s - --docker



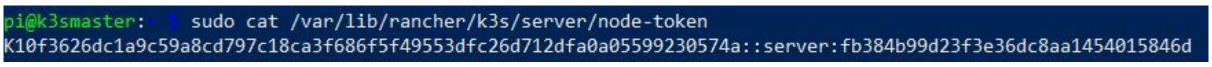
```
pi@k3smaster:~ $ curl -sfL https://get.k3s.io | sh -s - --docker
[INFO]  Finding release for channel stable
[INFO]  Using v1.25.6+k3s1 as release
[INFO]  Downloading hash https://github.com/k3s-io/k3s/releases/download/v1.25.6+k3s1/sha256sum-arm64.txt
[INFO]  Skipping binary downloaded, installed k3s matches hash
[INFO]  Skipping installation of SELinux RPM
[INFO]  Skipping /usr/local/bin/kubectl symlink to k3s, already exists
[INFO]  Skipping /usr/local/bin/crictl symlink to k3s, already exists
[INFO]  Skipping /usr/local/bin/ctr symlink to k3s, command exists in PATH at /usr/bin/ctr
[INFO]  Creating killall script /usr/local/bin/k3s-killall.sh
[INFO]  Creating uninstall script /usr/local/bin/k3s-uninstall.sh
[INFO]  env: Creating environment file /etc/systemd/system/k3s.service.env
[INFO]  systemd: Creating service file /etc/systemd/system/k3s.service
[INFO]  systemd: Enabling k3s unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s.service → /etc/systemd/system/k3s.service.
[INFO]  systemd: Starting k3s
```

- After successful run, this will start K3S service and it will automatically create few configurations file as well.
- Now check if the master node is ready using the command:
  sudo kubectl get nodes -o wide



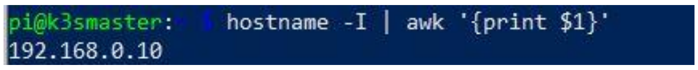
```
pi@k3smaster:~ $ sudo kubectl get nodes -o wide
NAME        STATUS   ROLES                  AGE    VERSION        INTERNAL-IP    EXTERNAL-IP   OS-IMAGE                         KERNEL-VERSION   CONTAINER-RUNTIME
k3smaster   Ready    control-plane,master   3m34s  v1.25.6+k3s1   192.168.0.10   <none>        Debian GNU/Linux 11 (bullseye)   5.15.61-v8+      docker://20.10.5+dfsg1
```

- To add agent or worker nodes, first we will generate token from master, this token will then be used by all agent nodes while creating the K3S cluster.
- Run below command and save the generated in a text editor:
  sudo cat /var/lib/rancher/k3s/server/node-token



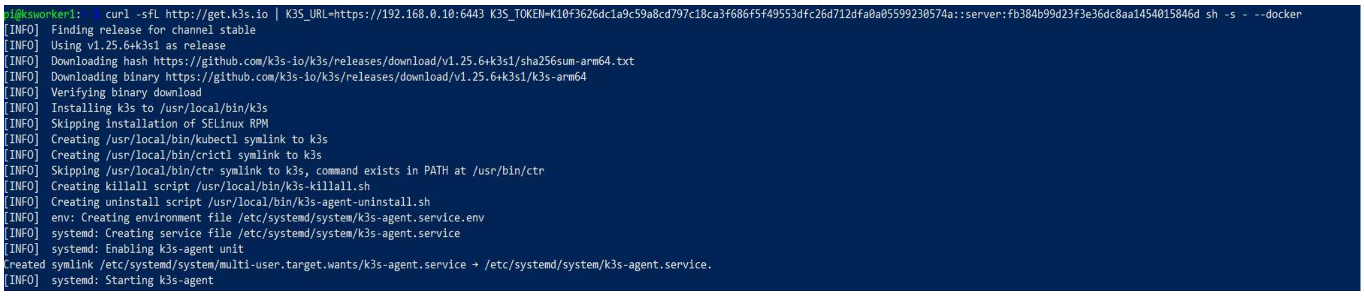
```
pi@k3smaster:~ $ sudo cat /var/lib/rancher/k3s/server/node-token
K10f3626dc1a9c59a8cd797c18ca3f686f5f49553dfc26d712dfa0a05599230574a::server:fb384b99d23f3e36dc8aa1454015846d
```

- Also check the Master IP using the command
  hostname -I | awk '{print $1}'



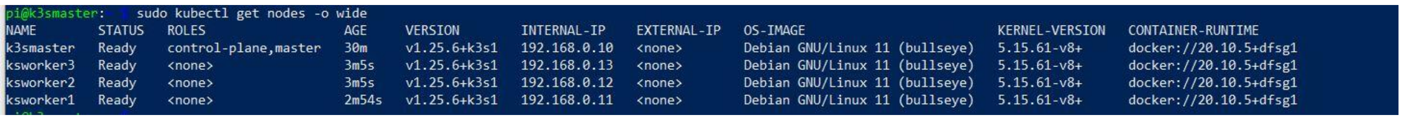
```
pi@k3smaster:~ $ hostname -I | awk '{print $1}'
192.168.0.10
```

- Now SSH each worker nodes and following command:
  curl -sfL https://get.k3s.io | K3S_URL=https://<Master_IP>:6443 K3S_TOKEN=<token_generated_from_master> sh -s - --docker



```
pi@ksworker1:~ $ curl -sfL http://get.k3s.io | K3S_URL=https://192.168.0.10:6443 K3S_TOKEN=K10f3626dc1a9c59a8cd797c18ca3f686f5f49553dfc26d712dfa0a05599230574a::server:fb384b99d23f3e36dc8aa1454015846d sh -s - --docker
[INFO]  Finding release for channel stable
[INFO]  Using v1.25.6+k3s1 as release
[INFO]  Downloading hash https://github.com/k3s-io/k3s/releases/download/v1.25.6+k3s1/sha256sum-arm64.txt
[INFO]  Downloading binary https://github.com/k3s-io/k3s/releases/download/v1.25.6+k3s1/k3s-arm64
[INFO]  Verifying binary download
[INFO]  Installing k3s to /usr/local/bin/k3s
[INFO]  Skipping installation of SELinux RPM
[INFO]  Creating /usr/local/bin/kubectl symlink to k3s
[INFO]  Creating /usr/local/bin/crictl symlink to k3s
[INFO]  Skipping /usr/local/bin/ctr symlink to k3s, command exists in PATH at /usr/bin/ctr
[INFO]  Creating killall script /usr/local/bin/k3s-killall.sh
[INFO]  Creating uninstall script /usr/local/bin/k3s-agent-uninstall.sh
[INFO]  env: Creating environment file /etc/systemd/system/k3s-agent.service.env
[INFO]  systemd: Creating service file /etc/systemd/system/k3s-agent.service
[INFO]  systemd: Enabling k3s-agent unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s-agent.service → /etc/systemd/system/k3s-agent.service.
[INFO]  systemd: Starting k3s-agent
```

- Now repeat this for all 3 worker nodes.
- Once done our K3S cluster is ready and we can check again in master node by using the command:
  sudo kubectl get nodes -o wide

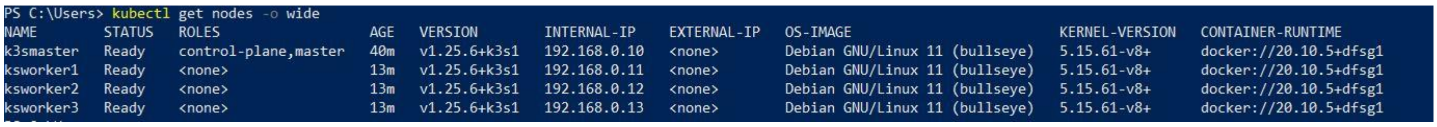


```
pi@k3smaster:~ $ sudo kubectl get nodes -o wide
NAME        STATUS   ROLES                  AGE    VERSION        INTERNAL-IP    EXTERNAL-IP   OS-IMAGE                         KERNEL-VERSION   CONTAINER-RUNTIME
k3smaster   Ready    control-plane,master   30m    v1.25.6+k3s1   192.168.0.10   <none>        Debian GNU/Linux 11 (bullseye)   5.15.61-v8+      docker://20.10.5+dfsg1
ksworker3   Ready    <none>                 3m5s   v1.25.6+k3s1   192.168.0.13   <none>        Debian GNU/Linux 11 (bullseye)   5.15.61-v8+      docker://20.10.5+dfsg1
ksworker2   Ready    <none>                 3m5s   v1.25.6+k3s1   192.168.0.12   <none>        Debian GNU/Linux 11 (bullseye)   5.15.61-v8+      docker://20.10.5+dfsg1
ksworker1   Ready    <none>                 2m54s  v1.25.6+k3s1   192.168.0.11   <none>        Debian GNU/Linux 11 (bullseye)   5.15.61-v8+      docker://20.10.5+dfsg1
```

5.3. Setting up kubectl on the local computer

This is an optional step which we followed and is not mandatory. By configuring the kubectl on our local machine, we can access our cluster locally and need not SSH every time.

- SSH into master and run the command:
  sudo cat /etc/rancher/k3s/k3s.yaml
- Copy the content of this file and into local file **kubeconfig.** In this file update the localhost IP to the Master node IP.
- Now the cluster should be accessible from our PC using the kubectl commands.
  kubectl get nodes -o wide



```
PS C:\Users> kubectl get nodes -o wide
NAME        STATUS   ROLES                  AGE   VERSION        INTERNAL-IP    EXTERNAL-IP   OS-IMAGE                         KERNEL-VERSION   CONTAINER-RUNTIME
k3smaster   Ready    control-plane,master   40m   v1.25.6+k3s1   192.168.0.10   <none>        Debian GNU/Linux 11 (bullseye)   5.15.61-v8+      docker://20.10.5+dfsg1
ksworker1   Ready    <none>                 13m   v1.25.6+k3s1   192.168.0.11   <none>        Debian GNU/Linux 11 (bullseye)   5.15.61-v8+      docker://20.10.5+dfsg1
ksworker2   Ready    <none>                 13m   v1.25.6+k3s1   192.168.0.12   <none>        Debian GNU/Linux 11 (bullseye)   5.15.61-v8+      docker://20.10.5+dfsg1
ksworker3   Ready    <none>                 13m   v1.25.6+k3s1   192.168.0.13   <none>        Debian GNU/Linux 11 (bullseye)   5.15.61-v8+      docker://20.10.5+dfsg1
```

6. REST API

The backend application is to save and retrieve data for images detected by the sensor node. This means that the system is designed to receive data from the sensor node, store it in a database or other persistent storage, and allow the retrieval of that data as needed.

The use of Java and Spring Boot framework in this system ensures that it is efficient, scalable and secure, making it well suited to handle large amounts of data and to provide a stable and reliable service.

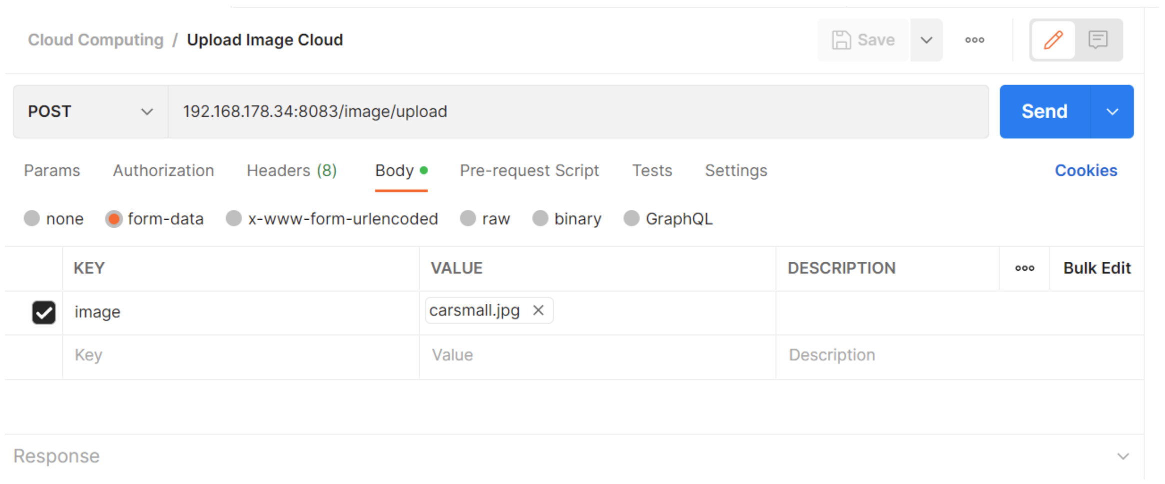
Technologies Used

- **Java**: The application is designed on Java programming language with version 11. The use of Java 11 provides access to a rich set of libraries and frameworks, allowing to create and scalable applications with ease.
- **Maven**: The application is also a Maven project, which means it is built and managed using the Apache Maven build automation tool. It provides a standardized way to manage dependencies, build the application and manage the development process.
- **Hibernate**: The system uses Hibernate, an Object-Relational Mapping (ORM) framework, to interact with the database. Hibernate provides a convenient way to map Java objects to database tables and vice versa, allowing the application to perform database operations without writing a raw SQL code.
- **Spring Boot**: The system is built using the Spring Boot Framework, version 2.7.5. It is a widely used framework for building web applications that provides a number of benefits, including ease of development and improved performance. With version 2.7.5, the developers will have access to the latest features and bug fixes, ensuring that the application is stable, efficient, and secure.
- **Docker**: A containerization technology used to deploy and run software applications in isolated environments, providing benefits such as isolation, portability, scalability, reproducibility, and automation.
- **Postgres**: A widely used and open-source relational database management system used to store and manage data.
- **REST API**: A standard for building web APIs that allows the backend application to interact with other systems over the internet.
- **Slack**: A communication channel to provide notification for the detected rat images.

APIs:

POST /image/upload

The goal of this REST API is to receive an image detected by the sensor node and save it in the database for the future retrieval. It accepts the file in the RequestParam as a Multipart file and persist it into postgres database. After saving the data, it is also responsible to call slack api to provide the notification details for the rat detection.

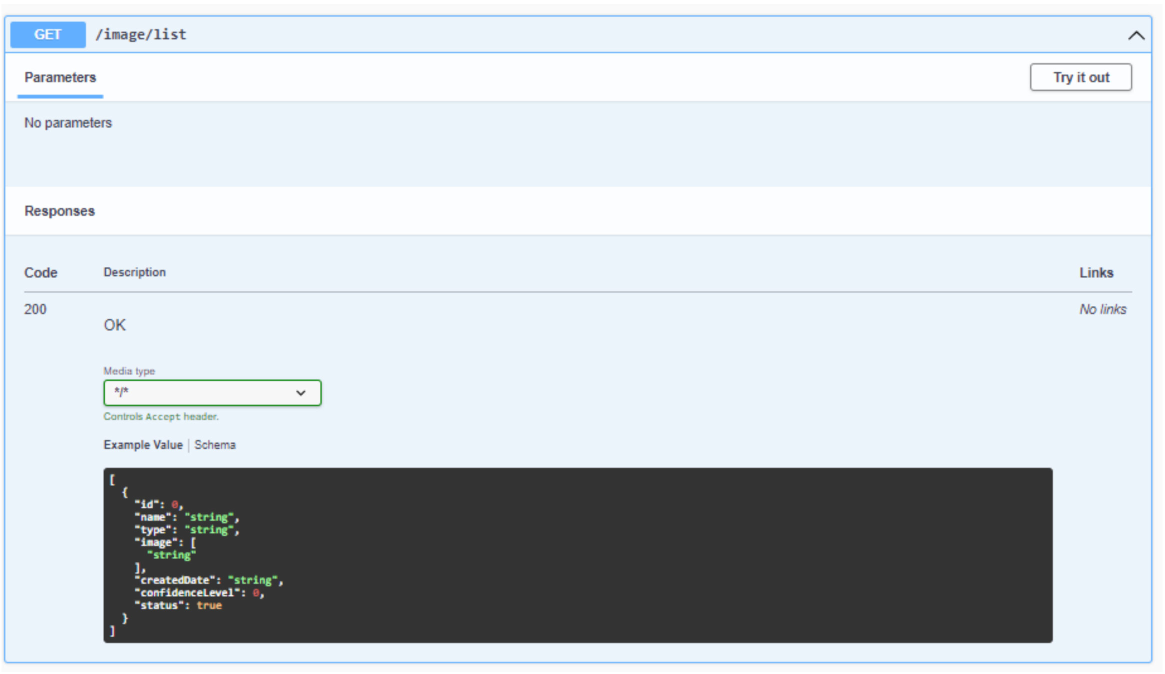


POST 192.168.178.34:8083/image/upload

	KEY	VALUE	DESCRIPTION
✓	image	carsmall.jpg	
	Key	Value	Description

GET/image/list

The purpose of this REST API is to provide the details of the saved images in the database to the client.

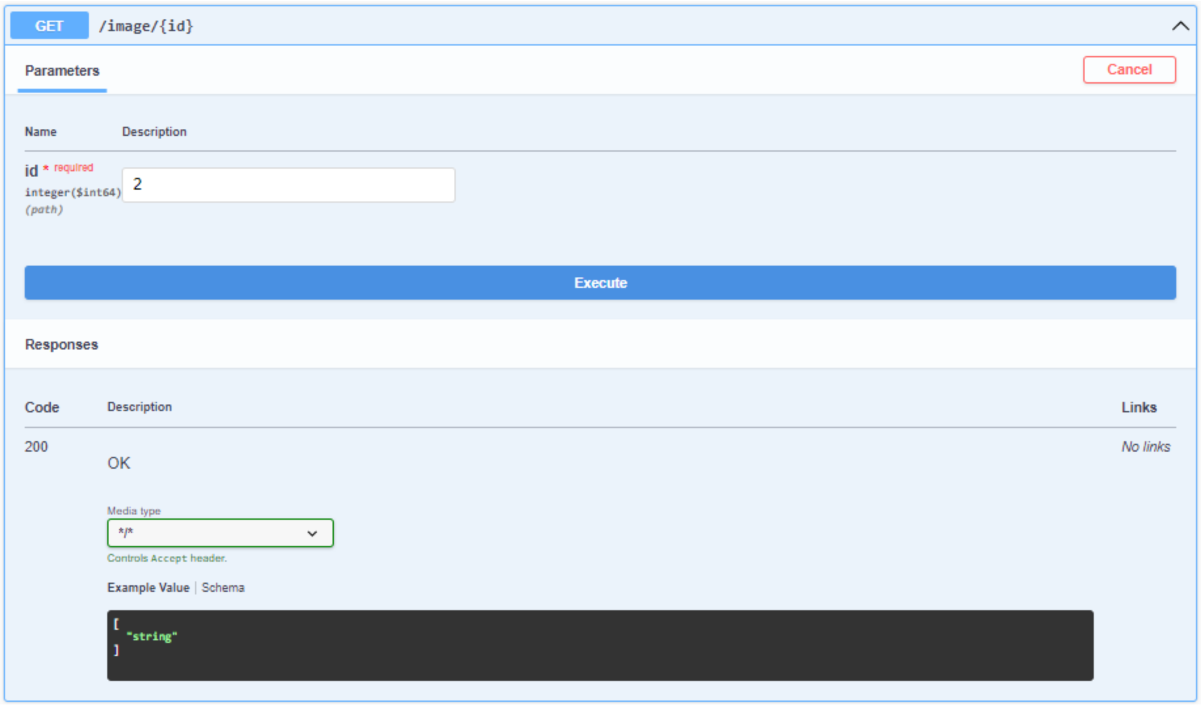


Code	Description	Links
200	OK	No links

```
[
  {
    "id": 0,
    "name": "string",
    "type": "string",
    "image": [
      "string"
    ],
    "createdDate": "string",
    "confidenceLevel": 0,
    "status": true
  }
]
```

GET/image/{id}

This API provides the actual detected image to the client. It accepts id of the image in the PathVariable for which the backend application executes the query to fetch the specified image in the database and provides it to the client side.



/image/{id}

Parameters

Name	Description
id * required integer($int64) (path)	2

Responses

Code	Description	Links
200	OK	No links

Media type: */*

Controls Accept header.

Example Value | Schema

```
[
  "string"
]
```

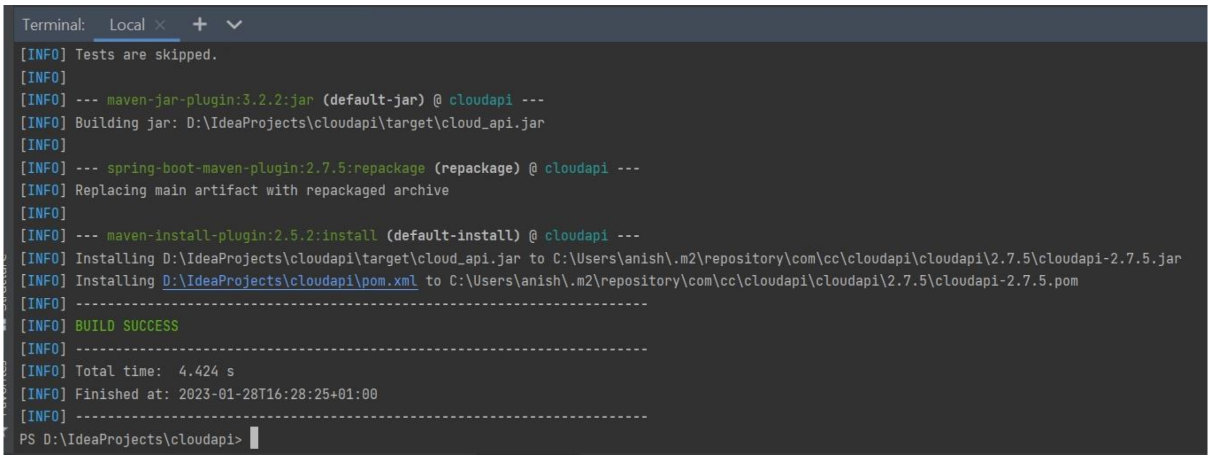
Local Docker Deployment

The backend application relies on the PostgresSQL container; thus, we employed a docker compose environment file to generate an image and execute the application within a docker container.

Steps in building application image and running in container.

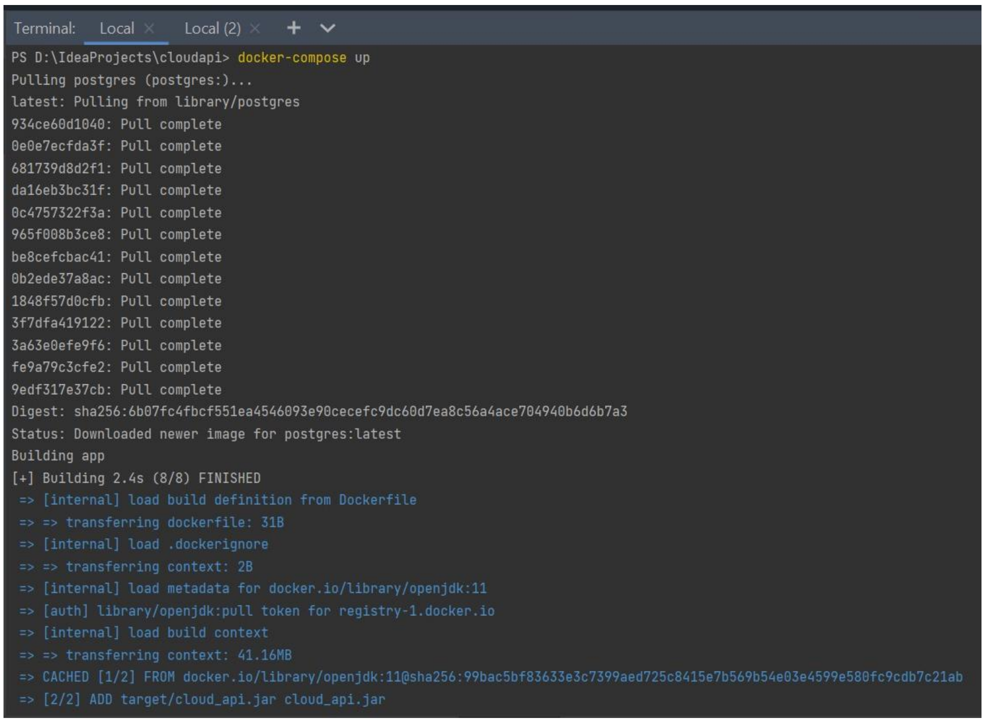
- From the code base root directory, creating an application jar with the command

mvn clean install -DskipTests



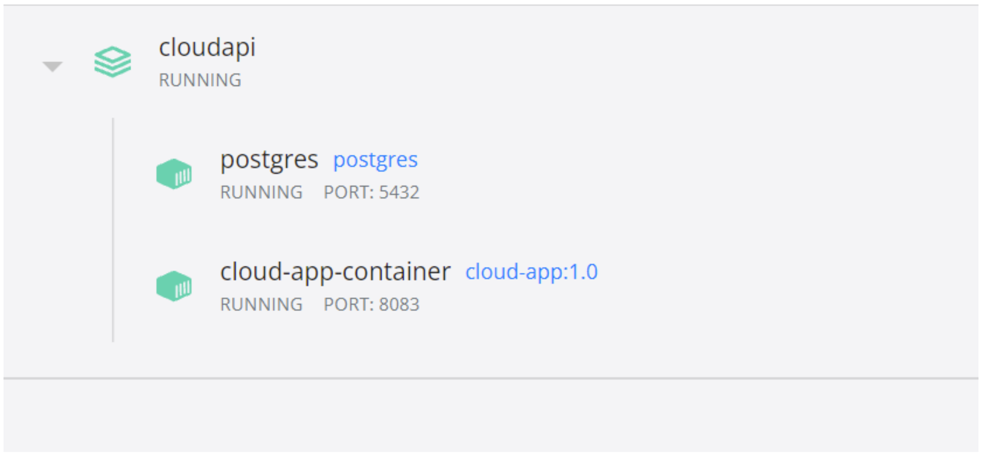
```
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-jar-plugin:3.2.2:jar (default-jar) @ cloudapi ---
[INFO] Building jar: D:\IdeaProjects\cloudapi\target\cloud_api.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.7.5:repackage (repackage) @ cloudapi ---
[INFO] Replacing main artifact with repackaged archive
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ cloudapi ---
[INFO] Installing D:\IdeaProjects\cloudapi\target\cloud_api.jar to C:\Users\anish\.m2\repository\com\cc\cloudapi\cloudapi\2.7.5\cloudapi-2.7.5.jar
[INFO] Installing D:\IdeaProjects\cloudapi\pom.xml to C:\Users\anish\.m2\repository\com\cc\cloudapi\cloudapi\2.7.5\cloudapi-2.7.5.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.424 s
[INFO] Finished at: 2023-01-28T16:28:25+01:00
[INFO] ------------------------------------------------------------------------
PS D:\IdeaProjects\cloudapi>
```

- Run the docker compose file with the command
  docker-compose up



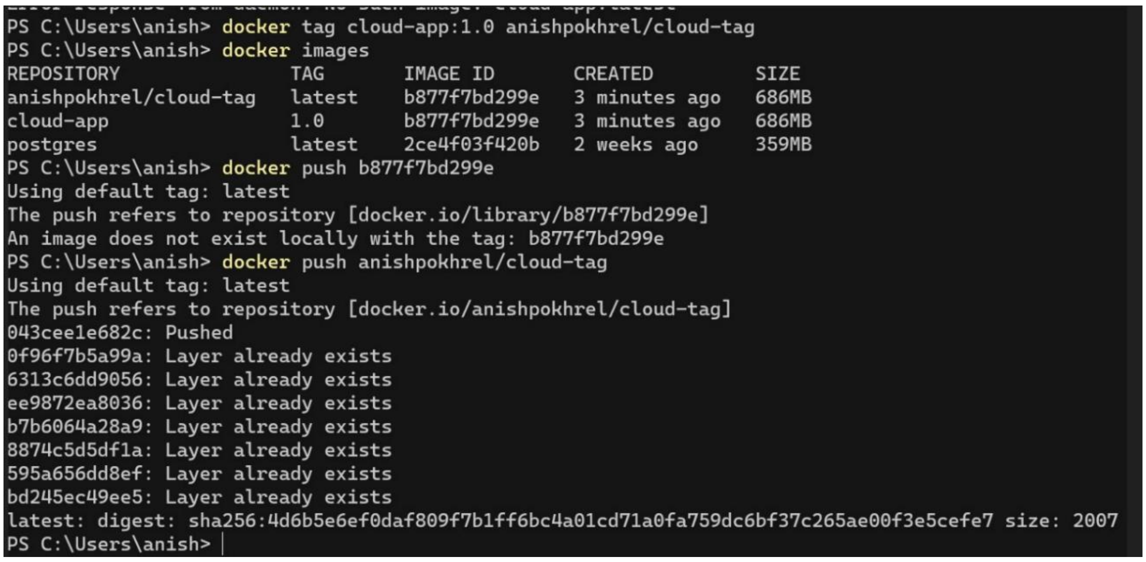
```
PS D:\IdeaProjects\cloudapi> docker-compose up
Pulling postgres (postgres:)...
latest: Pulling from library/postgres
934ce60d1040: Pull complete
0e0e7ecfda3f: Pull complete
681739d8d2f1: Pull complete
da16eb3bc31f: Pull complete
0c4757322f3a: Pull complete
965f008b3ce8: Pull complete
be8cefcbac41: Pull complete
0b2ede37a8ac: Pull complete
1848f57d0cfb: Pull complete
3f7dfa419122: Pull complete
3a63e0efe9f6: Pull complete
fe9a79c3cfe2: Pull complete
9edf317e37cb: Pull complete
Digest: sha256:6b07fc4fbcf551ea4546093e90cecefc9dc60d7ea8c56a4ace704940b6d6b7a3
Status: Downloaded newer image for postgres:latest
Building app
[+] Building 2.4s (8/8) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 31B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/openjdk:11
 => [auth] library/openjdk:pull token for registry-1.docker.io
 => [internal] load build context
 => => transferring context: 41.16MB
 => CACHED [1/2] FROM docker.io/library/openjdk:11@sha256:99bac5bf83633e3c7399aed725c8415e7b569b54e03e4599e580fc9cdb7c21ab
 => [2/2] ADD target/cloud_api.jar cloud_api.jar
```

- Postgres image and backend application will be created respectively and runs on the specified ports.

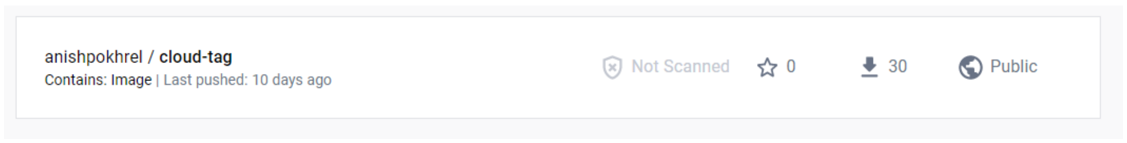


```
cloudapi
RUNNING
    postgres  postgres
    RUNNING   PORT: 5432
    cloud-app-container  cloud-app:1.0
    RUNNING   PORT: 8083
```

Finally, the private dockerhub repository was created to host the image of our own build application. The tag of the application image was created and pushed into the repository.



```
PS C:\Users\anish> docker tag cloud-app:1.0 anishpokhrel/cloud-tag
PS C:\Users\anish> docker images
REPOSITORY              TAG       IMAGE ID       CREATED         SIZE
anishpokhrel/cloud-tag  latest    b877f7bd299e   3 minutes ago   686MB
cloud-app               1.0       b877f7bd299e   3 minutes ago   686MB
postgres                latest    2ce4f03f420b   2 weeks ago     359MB
PS C:\Users\anish> docker push b877f7bd299e
Using default tag: latest
The push refers to repository [docker.io/library/b877f7bd299e]
An image does not exist locally with the tag: b877f7bd299e
PS C:\Users\anish> docker push anishpokhrel/cloud-tag
Using default tag: latest
The push refers to repository [docker.io/anishpokhrel/cloud-tag]
043cee1e682c: Pushed
0f96f7b5a99a: Layer already exists
6313c6dd9056: Layer already exists
ee9872ea8036: Layer already exists
b7b6064a28a9: Layer already exists
8874c5d5df1a: Layer already exists
595a656dd8ef: Layer already exists
bd245ec49ee5: Layer already exists
latest: digest: sha256:4d6b5e6ef0daf809f7b1ff6bc4a01cd71a0fa759dc6bf37c265ae00f3e5cefe7 size: 2007
PS C:\Users\anish> 
```



anishpokhrel / **cloud-tag**
Contains: Image | Last pushed: 10 days ago

Not Scanned · 0 · 30 · Public

7. Web Application

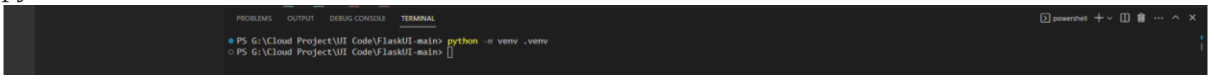
The frontend application will fetch data from REST API call and then display the corresponding data into the website. Flask is a micro web framework for Python that provides minimal functionality for building web applications; and Jinja is a fast, expressive, and extensible templating engine for Python that is used in Flask to render HTML templates. These tools, when combined, provide a scalable and flexible web development solution with an emphasis on simplicity and ease of use.

Technologies Used

- **Python:** The application is designed using Python programming language to provide a scalable and flexible web development solution with an emphasis on simplicity and ease of use.
- **Flask**: Flask is a micro web framework for Python that provides minimal functionality for building web applications,
- **Jinja**: Jinja is a fast, expressive, and extensible templating engine for Python that is used in Flask to render HTML templates
- **Docker:** Docker for deploying the application on different environments.

Local Setup

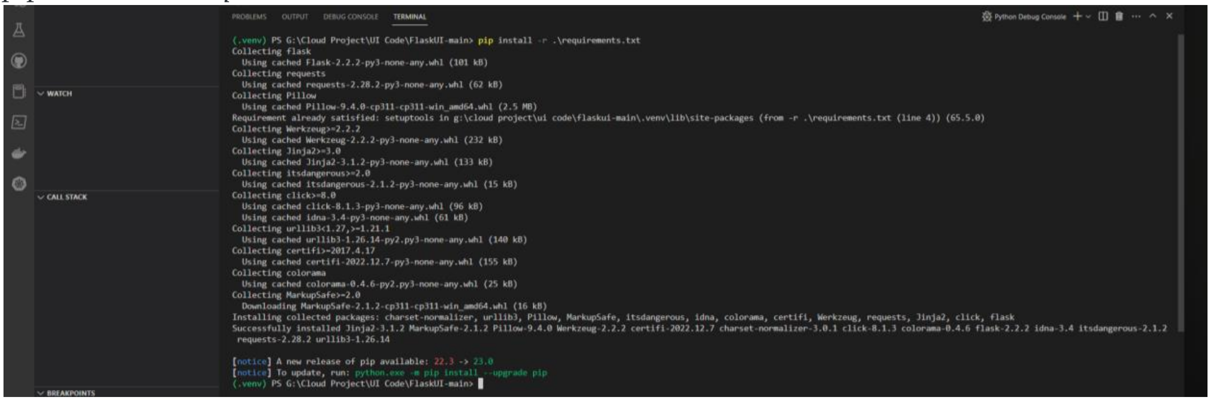
python -m venv .venv



```
PS G:\Cloud Project\UI Code\FlaskUI-main> python -m venv .venv
PS G:\Cloud Project\UI Code\FlaskUI-main> 
```

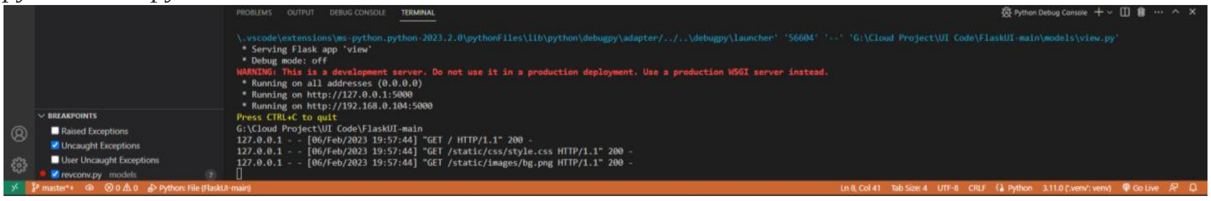
- Change the path to below to activate the created virtual environment and hit enter:
  .\.venv\Scripts\activate

- Create a requirements.txt file that will contains the list of libraries required for the application:
- Type below command to install the required python libraries and hit enter:
  pip install -r .\requirements.txt



```
(.venv) PS G:\Cloud Project\UI Code\FlaskUI-main> pip install -r .\requirements.txt
Collecting flask
  Using cached Flask-2.2.2-py3-none-any.whl (101 kB)
Collecting requests
  Using cached requests-2.28.2-py3-none-any.whl (62 kB)
Collecting Pillow
  Using cached Pillow-9.4.0-cp311-cp311-win_amd64.whl (2.5 MB)
Requirement already satisfied: setuptools in g:\cloud project\ui code\flaskui-main\.venv\lib\site-packages (from -r .\requirements.txt (line 4)) (65.5.0)
Collecting Werkzeug>=2.2.2
  Using cached Werkzeug-2.2.2-py3-none-any.whl (232 kB)
Collecting Jinja2>=3.0
  Using cached Jinja2-3.1.2-py3-none-any.whl (133 kB)
Collecting itsdangerous>=2.0
  Using cached itsdangerous-2.1.2-py3-none-any.whl (15 kB)
Collecting click>=8.0
  Using cached click-8.1.3-py3-none-any.whl (96 kB)
  Using cached idna-3.4-py3-none-any.whl (61 kB)
Collecting urllib3<1.27,>=1.21.1
  Using cached urllib3-1.26.14-py2.py3-none-any.whl (140 kB)
Collecting certifi>=2017.4.17
  Using cached certifi-2022.12.7-py3-none-any.whl (155 kB)
Collecting colorama
  Using cached colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-2.1.2-cp311-cp311-win_amd64.whl (16 kB)
Installing collected packages: charset-normalizer, urllib3, Pillow, MarkupSafe, itsdangerous, idna, colorama, certifi, Werkzeug, requests, Jinja2, click, flask
Successfully installed Jinja2-3.1.2 MarkupSafe-2.1.2 Pillow-9.4.0 Werkzeug-2.2.2 certifi-2022.12.7 charset-normalizer-3.0.1 click-8.1.3 colorama-0.4.6 flask-2.2.2 idna-3.4 itsdangerous-2.1.2
 requests-2.28.2 urllib3-1.26.14

[notice] A new release of pip available: 22.3 -> 23.0
[notice] To update, run: python.exe -m pip install --upgrade pip
(.venv) PS G:\Cloud Project\UI Code\FlaskUI-main> 
```

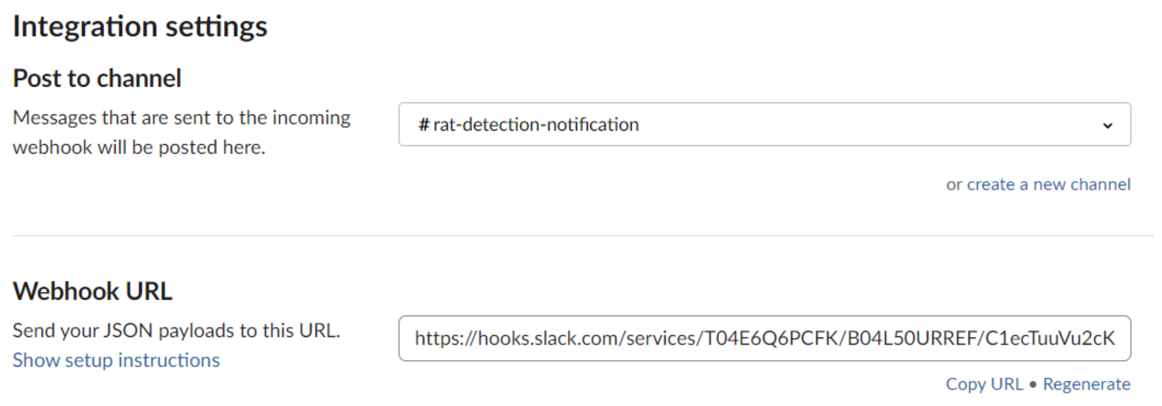
- Run the view.py file using command
  python view.py runserver



```
\.vscode\extensions\ms-python.python-2023.2.0\pythonFiles\lib\python\debugpy\adapter/../..\debugpy\launcher' '56604' '--' 'G:\Cloud Project\UI Code\FlaskUI-main\models\view.py'
 * Serving Flask app 'view'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.0.104:5000
Press CTRL+C to quit
G:\Cloud Project\UI Code\FlaskUI-main
127.0.0.1 - - [06/Feb/2023 19:57:44] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [06/Feb/2023 19:57:44] "GET /static/css/style.css HTTP/1.1" 200 -
127.0.0.1 - - [06/Feb/2023 19:57:44] "GET /static/images/bg.png HTTP/1.1" 200 -
```

8. Slack Notification Service

The notification service was implemented by integrating slack webhook in the backend application. For every image availability, the backend application saves the data to database and notifies the detection details by sending post request through slack webhook. In addition, we configured notification channel through slack web.


Integration settings

Post to channel

Messages that are sent to the incoming webhook will be posted here.

rat-detection-notification

Webhook URL

Send your JSON payloads to this URL.

https://hooks.slack.com/services/T04E6Q6PCFK/B04L50URREF/C1ecTuuVu2cK

9. Deployment

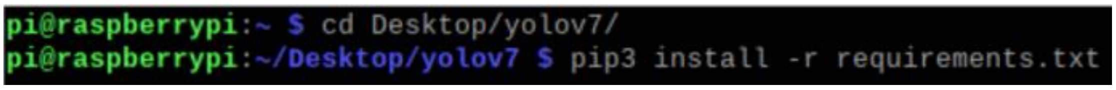
9.1. Deploying Machine Learning model on Sensor Node

Our machine learning model to detect rats is ready and now we can deploy it on our sensor node.

First, we need to download YOLOv7 folder from the Google Drive and copy it to a pen drive. We can clear certain files to reduce the size of the YOLOv7 folder such as, ***data\train & data\val*** folders containing training and validation data. After this is done, we insert the pen drive in the Raspberry Pi 4 USB port and copy the YOLOv7 folder onto the Desktop.

The YOLOv7 folder has a python script names ***detect.py*** which can be used to run detections. Before this can be done, we first need to install libraries required by this script on the sensor node, as we did in Google Colab environment.

Navigate to the folder where YOLOv7 folder was copied and run the command to install libraries as shown below,


```
pi@raspberrypi:~ $ cd Desktop/yolov7/
pi@raspberrypi:~/Desktop/yolov7 $ pip3 install -r requirements.txt
```

We are ready to run live detection now. Using the below command, we can start live rat detection,

python detect.py --weights best.pt --conf 0.5 --source 0 --no-trace --exist-ok

The command again takes few arguments which define the mode in which the detection is ran for ex.,

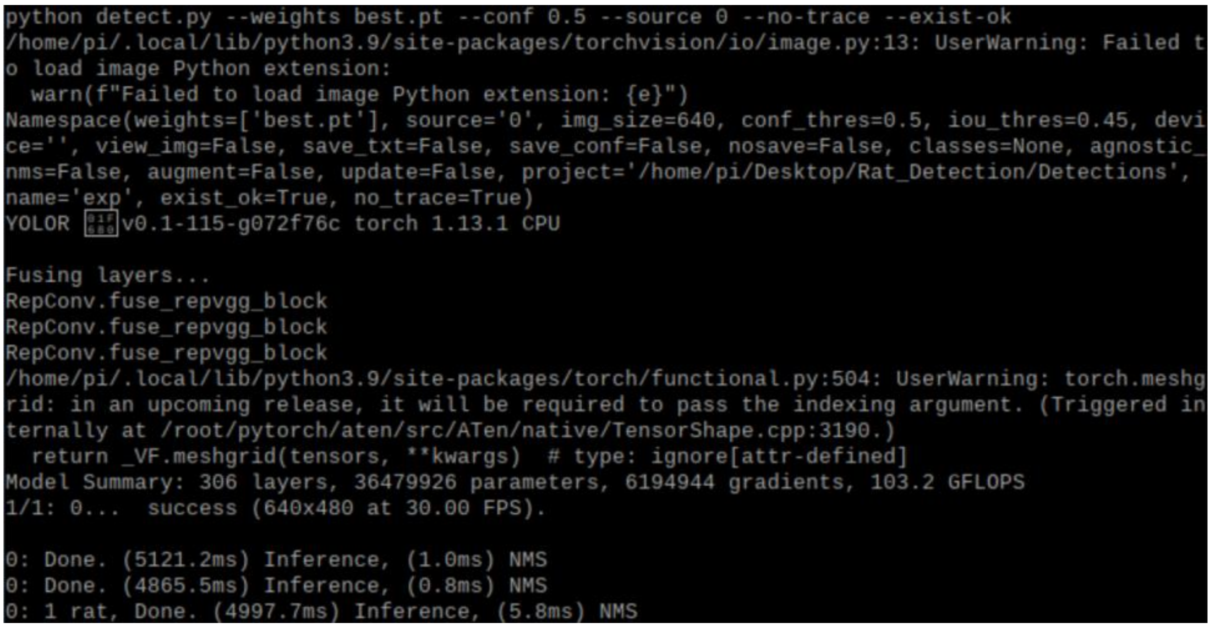
--weights: to point to the best.pt weight file,

--conf: to define the minimum confidence level for detections,

--source: to define the input source (0 is to access camera),

--no-trace: to disable tracing during detection, etc., The below screenshot shows verbose logs after detection is started. At the bottom we see the detections getting logged in the format,

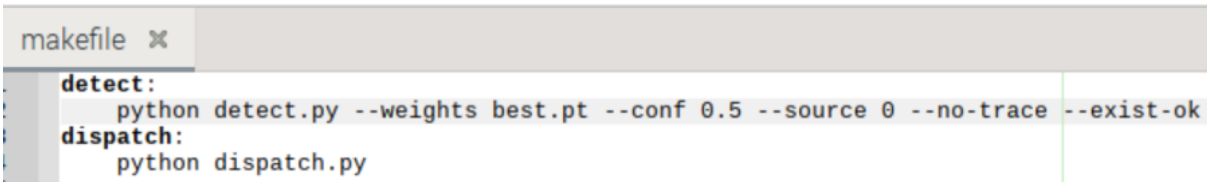
<source>: <rat count>



```
python detect.py --weights best.pt --conf 0.5 --source 0 --no-trace --exist-ok
/home/pi/.local/lib/python3.9/site-packages/torchvision/io/image.py:13: UserWarning: Failed t
o load image Python extension:
  warn(f"Failed to load image Python extension: {e}")
Namespace(weights=['best.pt'], source='0', img_size=640, conf_thres=0.5, iou_thres=0.45, devi
ce='', view_img=False, save_txt=False, save_conf=False, nosave=False, classes=None, agnostic_
nms=False, augment=False, update=False, project='/home/pi/Desktop/Rat_Detection/Detections',
name='exp', exist_ok=True, no_trace=True)
YOLOR v0.1-115-g072f76c torch 1.13.1 CPU

Fusing layers...
RepConv.fuse_repvgg_block
RepConv.fuse_repvgg_block
RepConv.fuse_repvgg_block
/home/pi/.local/lib/python3.9/site-packages/torch/functional.py:504: UserWarning: torch.meshg
rid: in an upcoming release, it will be required to pass the indexing argument. (Triggered in
ternally at /root/pytorch/aten/src/ATen/native/TensorShape.cpp:3190.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
Model Summary: 306 layers, 36479926 parameters, 6194944 gradients, 103.2 GFLOPS
1/1: 0...  success (640x480 at 30.00 FPS).

0: Done. (5121.2ms) Inference, (1.0ms) NMS
0: Done. (4865.5ms) Inference, (0.8ms) NMS
0: 1 rat, Done. (4997.7ms) Inference, (5.8ms) NMS
```

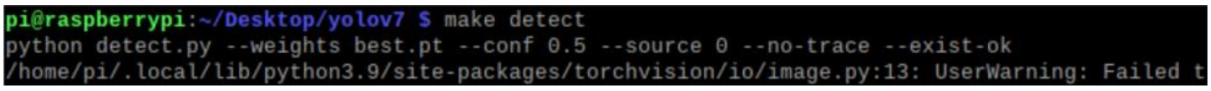
The detected rat images at the sensor node should be sent over to REST API running at the K3S cluster. To do this, we wrote another python script that dispatches images as soon as it is detected and saved on the disk in the folder ***/home/pi/Desktop/Rat_Detection/Detections***. The python script ***dispatch.py*** sends the detected rat images to the URL mentioned in the file ***/home/pi/Desktop/yolov7/URL.txt.*** If the dispatch was successful, the image is backed up to ***/home/pi/Desktop/Rat_Detection/Detections/Backup.*** If the dispatch was unsuccessful due to connectivity issues or some other reason, the dispatch script waits until connectivity is up again.

To automate the detection and dispatch processes, we wrote a makefile which eases the command calling process.

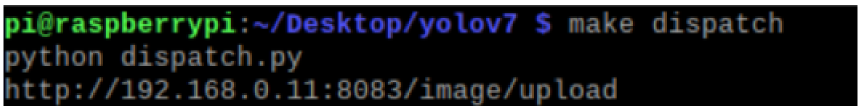


```
detect:
    python detect.py --weights best.pt --conf 0.5 --source 0 --no-trace --exist-ok
dispatch:
    python dispatch.py
```

Using this makefile, we can call the detection & dispatch scripts without passing any arguments as shown below,



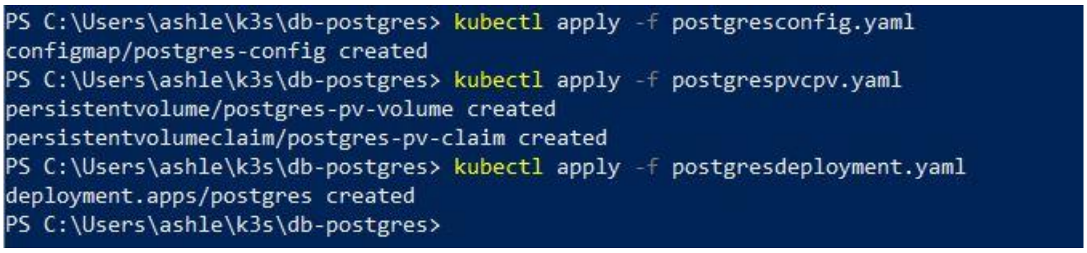
```
pi@raspberrypi:~/Desktop/yolov7 $ make detect
python detect.py --weights best.pt --conf 0.5 --source 0 --no-trace --exist-ok
/home/pi/.local/lib/python3.9/site-packages/torchvision/io/image.py:13: UserWarning: Failed t
```



```
pi@raspberrypi:~/Desktop/yolov7 $ make dispatch
python dispatch.py
http://192.168.0.11:8083/image/upload
```

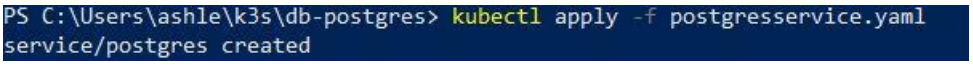
9.2. Setting up Postgres Database in the cluster

- The main advantage of having a PV is it creates a permanent storage and data will remain in the storage even if the pods are deleted.
- And with the help of PVC, users can request and consume PV resources.
- We have deployed the Postgres using Kubernetes manifest files:
  postgrespvcpv.yaml, *postgresconfig.yaml*, *postgresdeployment.yaml*



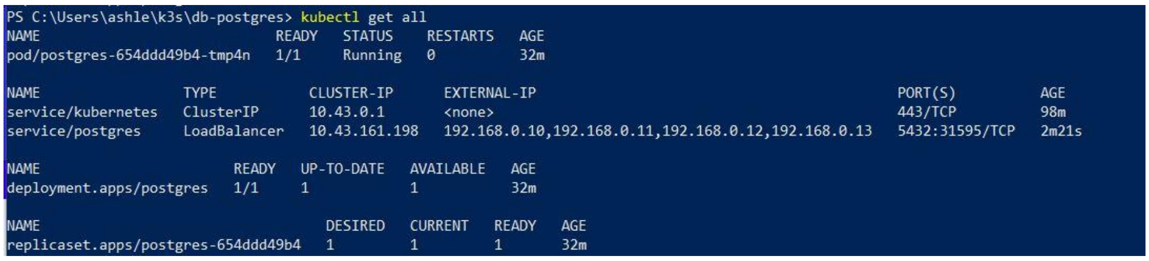
```
PS C:\Users\ashle\k3s\db-postgres> kubectl apply -f postgresconfig.yaml
configmap/postgres-config created
PS C:\Users\ashle\k3s\db-postgres> kubectl apply -f postgrespvcpv.yaml
persistentvolume/postgres-pv-volume created
persistentvolumeclaim/postgres-pv-claim created
PS C:\Users\ashle\k3s\db-postgres> kubectl apply -f postgresdeployment.yaml
deployment.apps/postgres created
PS C:\Users\ashle\k3s\db-postgres>
```

- Once deployment is done, the pod gets created and starts running, then we hosted Postgres service using service file: *postgresservice.yaml*.



```
PS C:\Users\ashle\k3s\db-postgres> kubectl apply -f postgresservice.yaml
service/postgres created
```

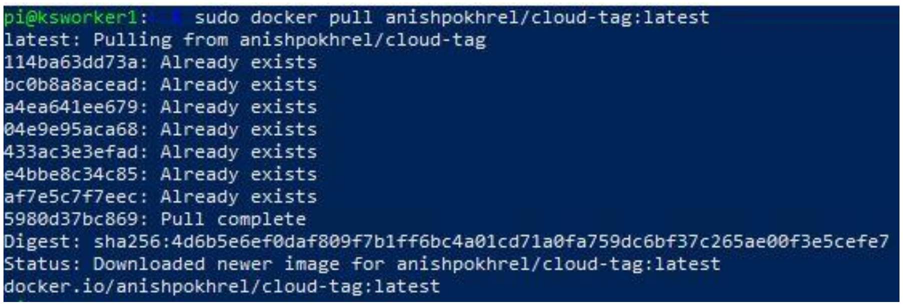
- As a result we were able to see all the services and deployments running the command: *kubectl get all*



```
PS C:\Users\ashle\k3s\db-postgres> kubectl get all
NAME                           READY   STATUS    RESTARTS   AGE
pod/postgres-654ddd49b4-tmp4n   1/1     Running   0          32m

NAME                 TYPE           CLUSTER-IP      EXTERNAL-IP                                               PORT(S)          AGE
service/kubernetes   ClusterIP      10.43.0.1       <none>                                                    443/TCP          98m
service/postgres     LoadBalancer   10.43.161.198   192.168.0.10,192.168.0.11,192.168.0.12,192.168.0.13   5432:31595/TCP   2m21s

NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/postgres   1/1     1            1           32m

NAME                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/postgres-654ddd49b4   1         1         1       32m
```

9.3. Deployment of Backend Application

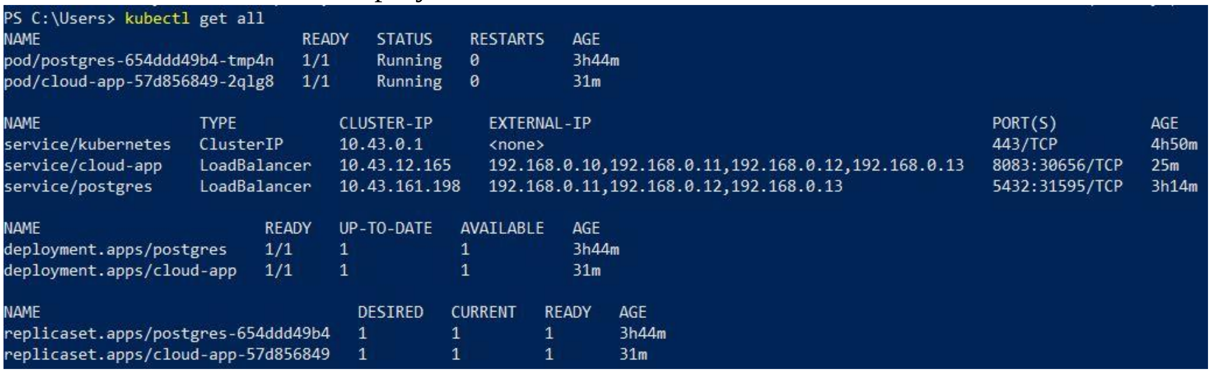
- After successful deployment of database service, we proceed to deploy backend application.
- The previously hosted docker images in private dockerhub repository were pulled in each worker nodes.



```
pi@ksworker1:   sudo docker pull anishpokhrel/cloud-tag:latest
latest: Pulling from anishpokhrel/cloud-tag
114ba63dd73a: Already exists
bc0b8a8acead: Already exists
a4ea641ee679: Already exists
04e9e95aca68: Already exists
433ac3e3efad: Already exists
e4bbe8c34c85: Already exists
af7e5c7f7eec: Already exists
5980d37bc869: Pull complete
Digest: sha256:4d6b5e6ef0daf809f7b1ff6bc4a01cd71a0fa759dc6bf37c265ae00f3e5cefe7
Status: Downloaded newer image for anishpokhrel/cloud-tag:latest
docker.io/anishpokhrel/cloud-tag:latest
```

- When the images are ready, we deploy and start the backend application using respective .yaml files.

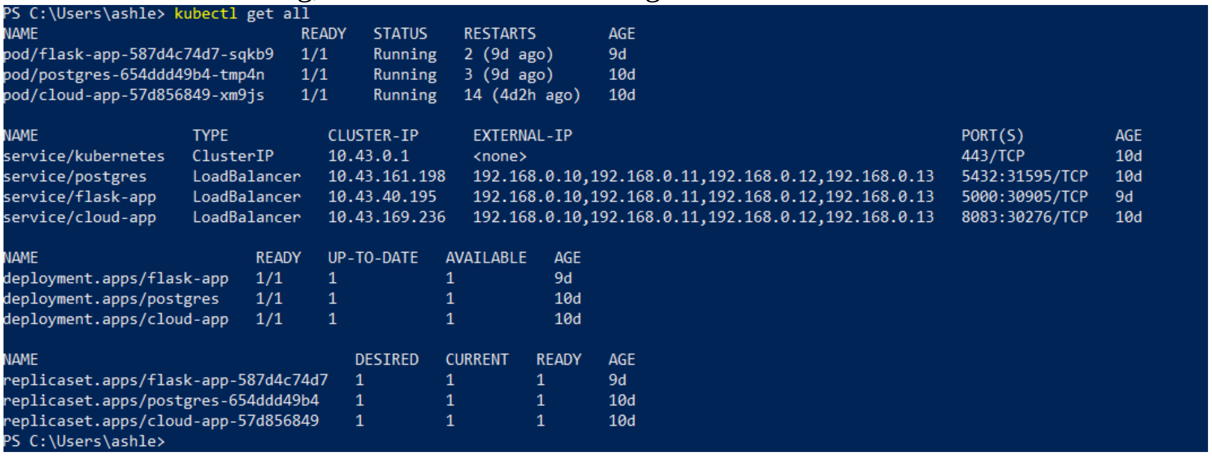
- We can see the successful deployment and service as shown



```
PS C:\Users> kubectl get all
NAME                           READY   STATUS    RESTARTS   AGE
pod/postgres-654ddd49b4-tmp4n  1/1     Running   0          3h44m
pod/cloud-app-57d856849-2qlg8  1/1     Running   0          31m

NAME                 TYPE           CLUSTER-IP      EXTERNAL-IP                                           PORT(S)          AGE
service/kubernetes   ClusterIP      10.43.0.1       <none>                                                443/TCP          4h50m
service/cloud-app    LoadBalancer   10.43.12.165    192.168.0.10,192.168.0.11,192.168.0.12,192.168.0.13   8083:30656/TCP   25m
service/postgres     LoadBalancer   10.43.161.198   192.168.0.11,192.168.0.12,192.168.0.13                5432:31595/TCP   3h14m

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/postgres     1/1     1            1           3h44m
deployment.apps/cloud-app    1/1     1            1           31m

NAME                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/postgres-654ddd49b4    1         1         1       3h44m
replicaset.apps/cloud-app-57d856849    1         1         1       31m
```

9.4. Deployment of Web Application

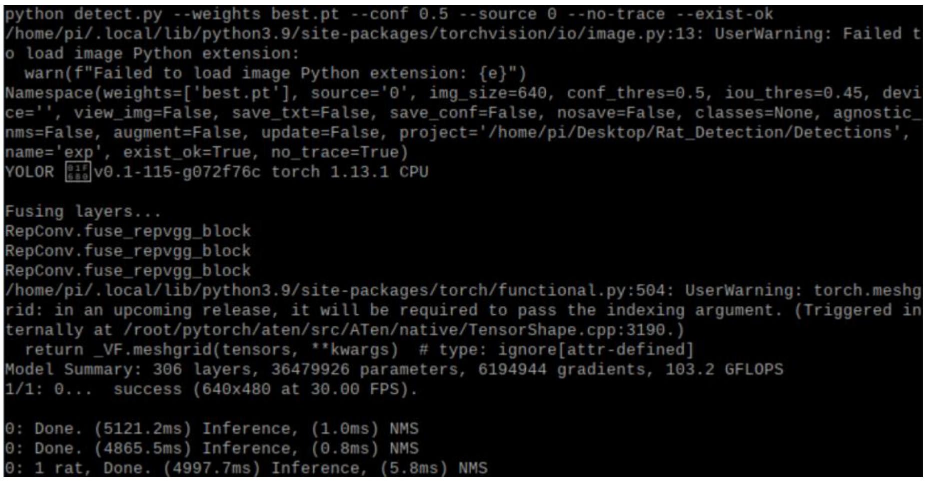
- Similarly, we deploy web applications by pulling the docker images into worker nodes and executing deployment and service files in master node for web application as well.
- After successful hosting, we can find the below logs as shown



```
PS C:\Users\ashle> kubectl get all
NAME                           READY   STATUS    RESTARTS       AGE
pod/flask-app-587d4c74d7-sqkb9 1/1     Running   2 (9d ago)     9d
pod/postgres-654ddd49b4-tmp4n  1/1     Running   3 (9d ago)     10d
pod/cloud-app-57d856849-xm9js  1/1     Running   14 (4d2h ago)  10d

NAME                 TYPE           CLUSTER-IP      EXTERNAL-IP                                           PORT(S)          AGE
service/kubernetes   ClusterIP      10.43.0.1       <none>                                                443/TCP          10d
service/postgres     LoadBalancer   10.43.161.198   192.168.0.10,192.168.0.11,192.168.0.12,192.168.0.13   5432:31595/TCP   10d
service/flask-app    LoadBalancer   10.43.40.195    192.168.0.10,192.168.0.11,192.168.0.12,192.168.0.13   5000:30905/TCP   9d
service/cloud-app    LoadBalancer   10.43.169.236   192.168.0.10,192.168.0.11,192.168.0.12,192.168.0.13   8083:30276/TCP   10d

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/flask-app    1/1     1            1           9d
deployment.apps/postgres     1/1     1            1           10d
deployment.apps/cloud-app    1/1     1            1           10d

NAME                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/flask-app-587d4c74d7   1         1         1       9d
replicaset.apps/postgres-654ddd49b4    1         1         1       10d
replicaset.apps/cloud-app-57d856849    1         1         1       10d
PS C:\Users\ashle>
```

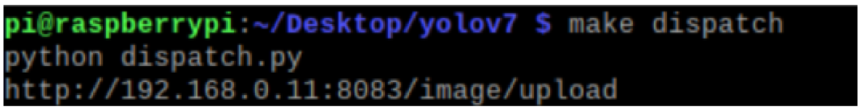
10. Results

After deploying all the services successfully, the cluster is ready for use. To check the final results, we run the live detection on the sensor node, the camera present will capture the live stream and whenever a rat is detected and is above the confidence level, the frame is captured and locally stored.



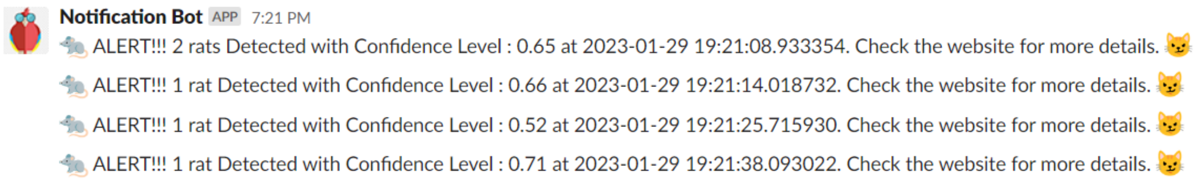
```
python detect.py --weights best.pt --conf 0.5 --source 0 --no-trace --exist-ok
/home/pi/.local/lib/python3.9/site-packages/torchvision/io/image.py:13: UserWarning: Failed t
o load image Python extension:
  warn(f"Failed to load image Python extension: {e}")
Namespace(weights=['best.pt'], source='0', img_size=640, conf_thres=0.5, iou_thres=0.45, devi
ce='', view_img=False, save_txt=False, save_conf=False, nosave=False, classes=None, agnostic_
nms=False, augment=False, update=False, project='/home/pi/Desktop/Rat_Detection/Detections',
name='exp', exist_ok=True, no_trace=True)
YOLOR ▯v0.1-115-g072f76c torch 1.13.1 CPU

Fusing layers...
RepConv.fuse_repvgg_block
RepConv.fuse_repvgg_block
RepConv.fuse_repvgg_block
/home/pi/.local/lib/python3.9/site-packages/torch/functional.py:504: UserWarning: torch.meshg
rid: in an upcoming release, it will be required to pass the indexing argument. (Triggered in
ternally at /root/pytorch/aten/src/ATen/native/TensorShape.cpp:3190.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
Model Summary: 306 layers, 36479926 parameters, 6194944 gradients, 103.2 GFLOPS
1/1: 0...  success (640x480 at 30.00 FPS).

0: Done. (5121.2ms) Inference, (1.0ms) NMS
0: Done. (4865.5ms) Inference, (0.8ms) NMS
0: 1 rat, Done. (4997.7ms) Inference, (5.8ms) NMS
```

The stored images are dispatched to the REST API as and when there are detections.



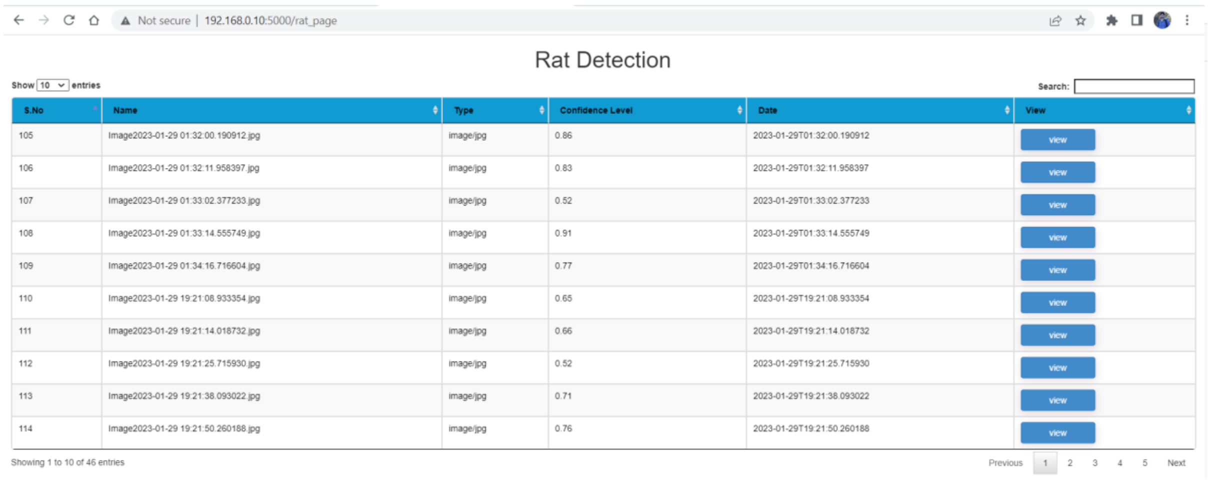
```
pi@raspberrypi:~/Desktop/yolov7 $ make dispatch
python dispatch.py
http://192.168.0.11:8083/image/upload
```

The images received by the REST API are stored in Postgres database and a Slack notification is generated.



```
Notification Bot APP 7:21 PM
ALERT!!! 2 rats Detected with Confidence Level : 0.65 at 2023-01-29 19:21:08.933354. Check the website for more details.
ALERT!!! 1 rat Detected with Confidence Level : 0.66 at 2023-01-29 19:21:14.018732. Check the website for more details.
ALERT!!! 1 rat Detected with Confidence Level : 0.52 at 2023-01-29 19:21:25.715930. Check the website for more details.
ALERT!!! 1 rat Detected with Confidence Level : 0.71 at 2023-01-29 19:21:38.093022. Check the website for more details.
```

The web application can be accessed to see all the history of rat detection images along with their timestamp and confidence level.

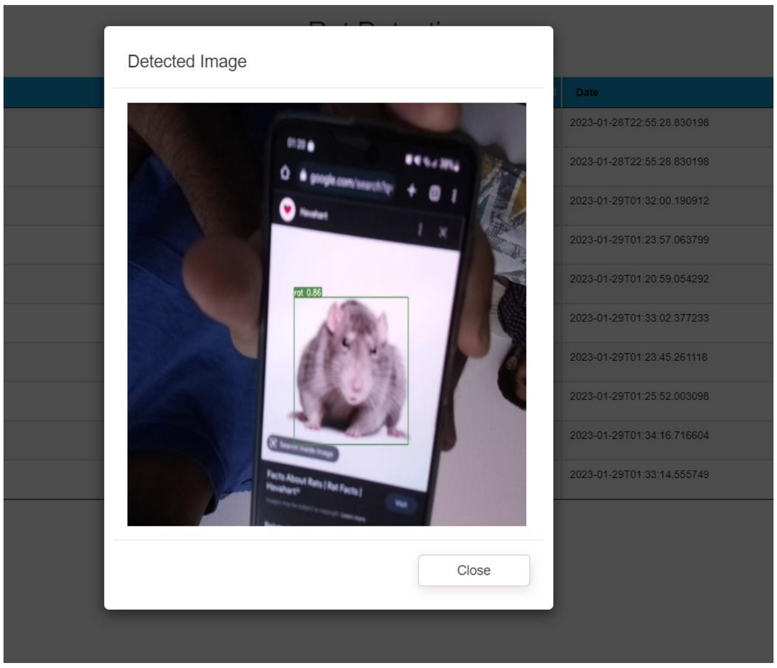


Rat Detection

Show 10 entries   Search:

S.No	Name	Type	Confidence Level	Date	View
105	Image2023-01-29 01:32:00.190912.jpg	image/jpg	0.86	2023-01-29T01:32:00.190912	view
106	Image2023-01-29 01:32:11.958397.jpg	image/jpg	0.83	2023-01-29T01:32:11.958397	view
107	Image2023-01-29 01:33:02.377233.jpg	image/jpg	0.52	2023-01-29T01:33:02.377233	view
108	Image2023-01-29 01:33:14.555749.jpg	image/jpg	0.91	2023-01-29T01:33:14.555749	view
109	Image2023-01-29 01:34:16.716604.jpg	image/jpg	0.77	2023-01-29T01:34:16.716604	view
110	Image2023-01-29 19:21:08.933354.jpg	image/jpg	0.65	2023-01-29T19:21:08.933354	view
111	Image2023-01-29 19:21:14.018732.jpg	image/jpg	0.66	2023-01-29T19:21:14.018732	view
112	Image2023-01-29 19:21:25.715930.jpg	image/jpg	0.52	2023-01-29T19:21:25.715930	view
113	Image2023-01-29 19:21:38.093022.jpg	image/jpg	0.71	2023-01-29T19:21:38.093022	view
114	Image2023-01-29 19:21:50.260188.jpg	image/jpg	0.76	2023-01-29T19:21:50.260188	view

Showing 1 to 10 of 46 entries   Previous 1 2 3 4 5 Next



Date
2023-01-28T22:55:28.830198
2023-01-28T22:55:28.830198
2023-01-29T01:32:00.190912
2023-01-29T01:23:57.063799
2023-01-29T01:20:59.054292
2023-01-29T01:33:02.377233
2023-01-29T01:23:45.261118
2023-01-29T01:25:52.003098
2023-01-29T01:34:16.716604
2023-01-29T01:33:14.555749

11. References

1. https://medium.com/thinkport/how-to-build-a-raspberry-pi-kubernetes-cluster-with-k3s-76224788576c
2. https://www.containiq.com/post/deploy-postgres-on-kubernetes
3. https://github.com/WongKinYiu/yolov7
4. https://medium.com/augmented-startups/yolov7-training-on-custom-data-b86d23e6623
5. https://www.analyticsvidhya.com/blog/2022/08/how-to-train-a-custom-object-detection-model-with-yolov7/
6. https://www.raspberrypi.com/documentation/computers/camera_software.html
7. https://www.raspberrypi.com/documentation/computers/configuration.html
8. https://www.raspberrypi.com/documentation/computers/os.html
9. https://docs.docker.com/docker-hub/
10. https://www.baeldung.com/rest-with-spring-course
11. https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3
12. https://flask.palletsprojects.com/en/2.2.x/quickstart/
13. https://www.kite.com/blog/python/flask-tutorial/
14. https://kubernetes.io/