Distributed System for Person Identification Using IoT, Edge, and Cloud Layers

1st Marcel Auer, 2nd Roman W. Jarz
3rd Gabriel Kofler, 4th Andreas Voppichler

January 22, 2025

1 Introduction

In this project, we aim to identify whether a person is "known" or "unknown" using a distributed system. The system simulates 5 IoT devices, an 2 edge devices, and a cloud. The IoT devices send a continuous stream of frames. If a person is identified as unknown, it triggers an alarm on the corresponding IoT device.

2 System Architecture

Our system simulates IoT devices by loading videos from a compressed dataset. These devices are EC2 containers capable of receiving messages and sending images. They forward images at defined intervals to the Amazon Simple-Message-Queue (SQS), which distributes them evenly across two EC2 edge instances.

At the edge, the images are filtered and forwarded to the cloud layer, which consists of a Lambda service. This service communicates with AWS Rekognition to identify the person. If the person is identified as unknown, it sends a message via SQS to the edge, which then forwards it to the IoT containers to trigger an alarm.

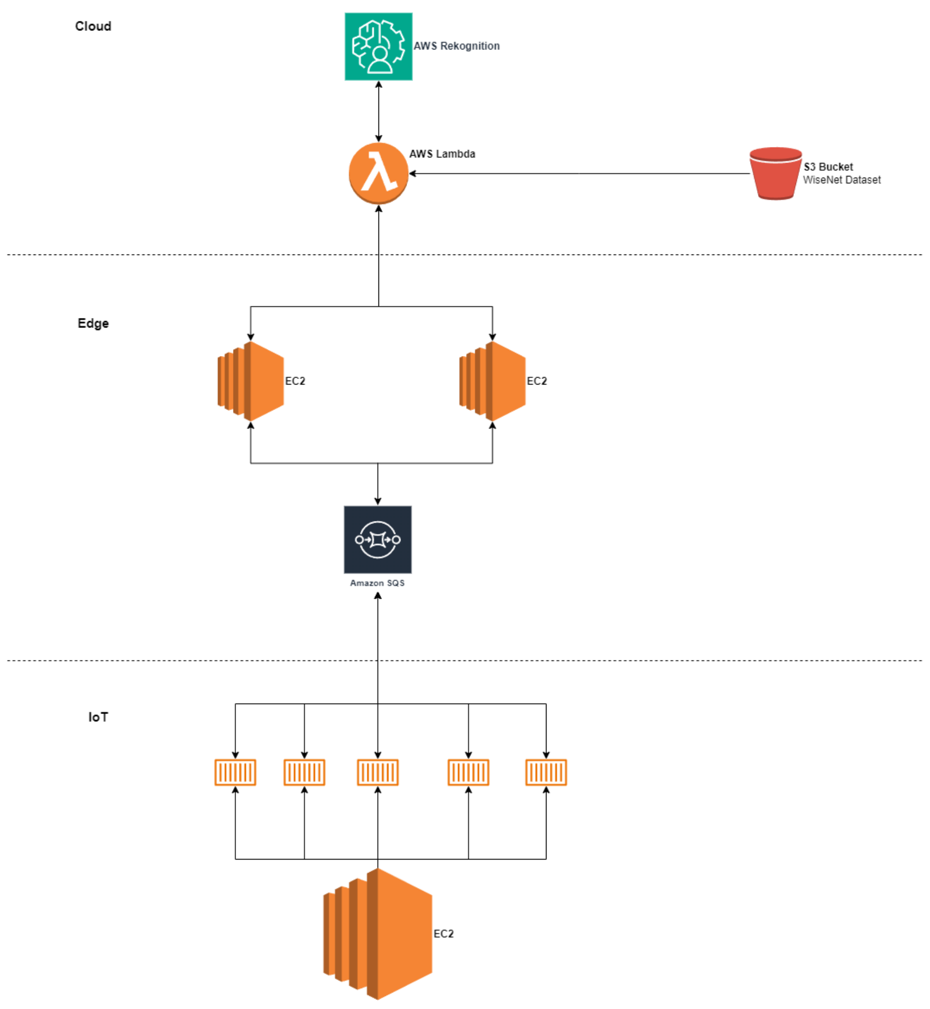


Figure 1: System architecture diagram.

3 Implementation Details

3.1 IoT Layer

We started by compressing the WiseNet-dataset to 1/10 of the original size (170 MB). This was done by using a bash file see algorithm 1.

Algorithm 1 Bash compression of .avi files using ffmpeg

```
Input: Folders containing ‘.avi‘ files
Output: Compressed ‘.avi‘ files
for each folder in */ do
  cd $folder
  for each file in *.avi do
    ffmpeg -i $file -vcodec libx264 -crf 28 compressed_$file
  end for
  cd ..
end for
```

Then we started to implement the code to pick an image from the compressed video set, with the openCV library. Made it multithreaded, so one thread picks and sends images, and one thread listens for an alarm notification. The messages are forwarded via a message queue. The JSON message contains the ID of each container and the image encoded in bytes. At first the encoding exceeded sometimes the 256 KB limit of a SQS message. Now a more efficient sending method is used, the cv2 encoding happens with a 90% quality and gets then converted to a json message with base64. This reduced the size of each message by approximately 25%.

Each of the 5 IoT-containers has the same AMI image including the whole dataset. Also, they all have the same Python file with different environment variables (id, video set, ...).

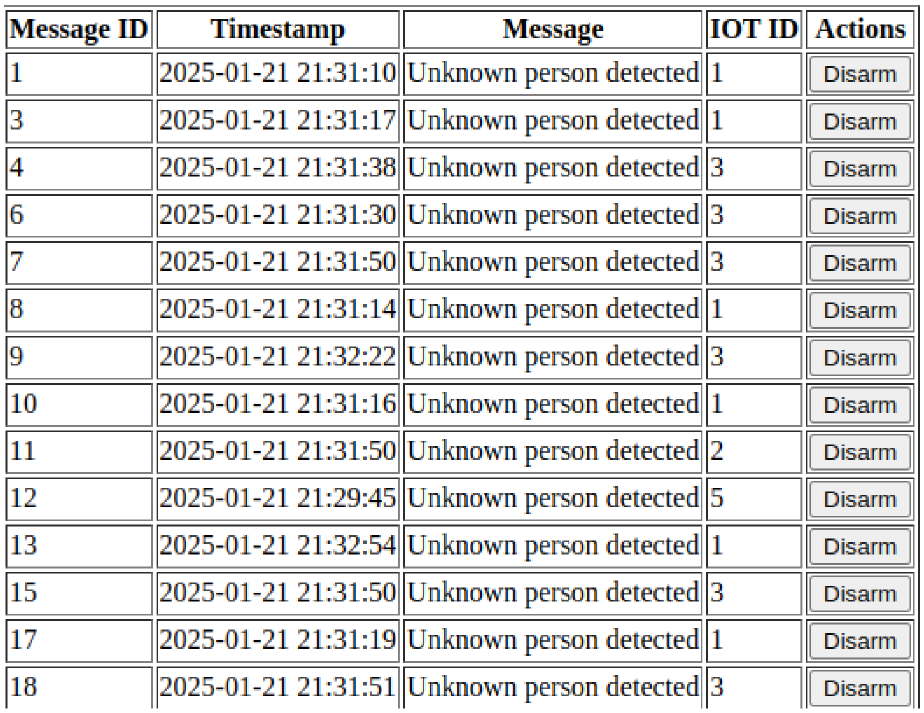
3.1.1 Setup of IoT Containers on EC2

We created a bash script in order to build the docker image and save it locally in a folder, which we upload with Terraform. We configured Terraform so that it will create an EC2 instance with LabInstanceProfile and upload the docker container for IoT as iot.tar (contains Python run script and the compressed sets of videos) file, in addition to the docker-compose file. To automate the process of creating the EC2 instance even further, we created a second bash file start.sh, which applies Terraform to create the instance uploads the files and installs all the dependencies on the EC2 for docker and docker-compose. Then it will automatically load the docker image from the uploaded iot.tar file, and with the help of docker-compose, we start 5 containers of this image each having a different environment variable to simulate the 5 IoT devices. To set everything up faster, we stopped the running containers and created a snapshot of the instance.

We then created an Amazon machine image (AMI), so that it is possible to load the working EC2 instance much faster. After creating the EC2 instance with the AMI, it is just necessary to choose LabInstanceProfile as IAM-Role and to start the containers with the command ”docker-compose up -d”.

3.1.2 Alarm Service

Upon detecting an unknown person the alarm message gets forwarded from the Cloud to the Edge and from there the SQS_queue forwards it to the corresponding IOT-device. The received alarm message gets displayed on localhost. In order to open the website it is necessary to execute the app.py file in the web-alarms folder. The alarms are listed in a simple table and can be disarmed by clicking on the button. Every alarm message has an unique ID and an associated timestamp.



Message ID	Timestamp	Message	IOT ID	Actions
1	2025-01-21 21:31:10	Unknown person detected	1	Disarm
3	2025-01-21 21:31:17	Unknown person detected	1	Disarm
4	2025-01-21 21:31:38	Unknown person detected	3	Disarm
6	2025-01-21 21:31:30	Unknown person detected	3	Disarm
7	2025-01-21 21:31:50	Unknown person detected	3	Disarm
8	2025-01-21 21:31:14	Unknown person detected	1	Disarm
9	2025-01-21 21:32:22	Unknown person detected	3	Disarm
10	2025-01-21 21:31:16	Unknown person detected	1	Disarm
11	2025-01-21 21:31:50	Unknown person detected	2	Disarm
12	2025-01-21 21:29:45	Unknown person detected	5	Disarm
13	2025-01-21 21:32:54	Unknown person detected	1	Disarm
15	2025-01-21 21:31:50	Unknown person detected	3	Disarm
17	2025-01-21 21:31:19	Unknown person detected	1	Disarm
18	2025-01-21 21:31:51	Unknown person detected	3	Disarm

Figure 2: Overview of Alarm Service Website

3.2 Edge Layer

Before we finished the edge layer, we decided to write a local version. Which has the basic logic of the IoT and Edge. With this local version, we could have a look at the results of the object detection model and we could also check if the encoding and decoding between IoT and edge works properly.

We then started to implement a multithreaded edge layer. At first we made a version where each filtered image calls the cloud layer, but this led to too many lambda invocations. Which then ended into our accounts getting banned. Now it is based on the consumer-producer pattern. One thread accesses the message queue, which then removes the message and decodes it back to an image. This then gets processed through an already pre-trained object detection model (ultralytics - YOLO). If the model detects a person with a confidence over 60% it crops the image down to the person that is detected. This saves a lot of data transfer, because we only want to recognize persons and the rest of the image is irrelevant. Now it forwards this cropped image towards a queue where one of 3 consumer-threads, pops the image encodes it and sends it to the cloud. The consumer-thread waits till a message comes back from the cloud and either sends an alarm to the IoT layer or prints that a known person is detected. Afterwards, this thread picks a new image from the queue if one exists or waits.

We had problems with the model, since we were not able to install the ultralytics library on the ec2 instance. Therefore, we had to remove that library, which was achieved by using our local version again. We had the files for the model's weights, architecture configuration, and the target labels, so we used OpenCV to map these configurations onto a CV2 neural network, which achieved similar results.

3.2.1 Setup of Edge Instances on EC2

We set up a bash script to execute the Terraform-file and to install all the required dependencies.This includes libraries, model weights and config. The Terraform-file automatically creates 2 EC2-instances with the same configuration which have t2.micro capacities and the LabInstanceProfile role. The EC2 instances can be accessed via ssh which requires an authentication-key named "EC2-key.pem" in our case.

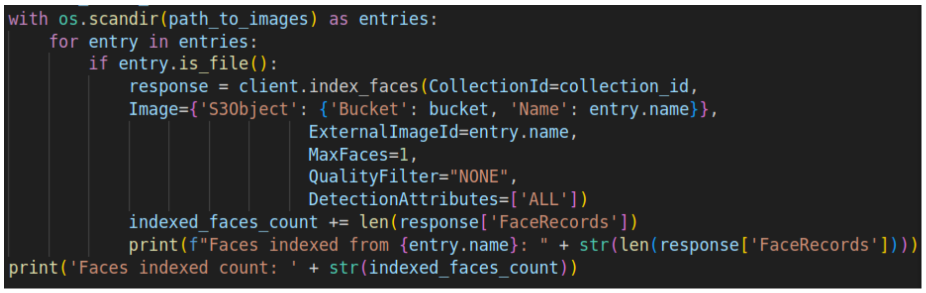
3.3 Cloud

3.3.1 Data Container

To save the known persons we need some kind of data container. We decided to use aws collections, because they synergize well with the rest of the aws recognition. We also saved the images in an s3 bucket, because aws collection doesn't save the images itself.
For the collection and the s3 bucket we used a script, which is executed once for the initialization and can be used to update the dataset with new images. As seen in figure 3 we use the index-faces function for the images from the s3 bucket. We use the name of the image as the ID, to avoid duplicated uploads once the collection is initialized.

3.3.2 Lambda Function

The lambda function is the main part of the cloud layer, it gets invoked by the edge layer and does the face recognition. The lambda function itself is pretty simple first we get a base64 encoded image from the edge layer, then we decode it and put it in the "search_faces_by_image" function of the boto3.client-library. The quality filter was set to none because a lot of images had bad quality and



```
with os.scandir(path_to_images) as entries:
    for entry in entries:
        if entry.is_file():
            response = client.index_faces(CollectionId=collection_id,
            Image={'S3Object': {'Bucket': bucket, 'Name': entry.name}},
                                ExternalImageId=entry.name,
                                MaxFaces=1,
                                QualityFilter="NONE",
                                DetectionAttributes=['ALL'])
            indexed_faces_count += len(response['FaceRecords'])
            print(f"Faces indexed from {entry.name}: " + str(len(response['FaceRecords'])))
print('Faces indexed count: ' + str(indexed_faces_count))
```

Figure 3: Initializing the collection with the known persons

the filter would have filtered out a lot of them. The ”FaceMatchThreshold” was set to 70 instead of the default value of 80, because we wanted to avoid false negatives due to bad quality/bad camera angle. The practical testing showed that this didn’t make much of a difference, because the matching was most of the time either above 90 % or below 50 % and only some edge cases were affected by this change.
The lambda function returns four status codes, ”known” for persons who are known, ”unknown” if the person is not found in the collection, ”no_face” if the image doesn’t contain a face, and ”error” if the image was not decoded correctly or recognition throws an internal error. Testing showed that ”no_face” is by far the most common status code, because the edge layer just filters for persons, but there are a lot of images from behind or from the side, which don’t contain a face. So we decided to treat the ”no_face” status code as ”known” persons because otherwise, we would get a lot of unnecessary false alarms.
For further improvements, it would make sense to save this image temporarily, if we need them in potential future investigations. Furthermore, we could use the ”detect_faces” function to sort out images, instead of using the exception thrown.

4 Evaluation

We evaluated the system by measuring the time of each layer. This was done under different stress levels, sending frames in an interval of 2 seconds per frame up to 0.1 seconds per frame. Each time 5000 messages were sent and processed. Therefore we had a whole bunch of data to process. The system could keep up with 0.5 frames per second, but with 0.25 frames per second the queue from iot to edge started to fill up.

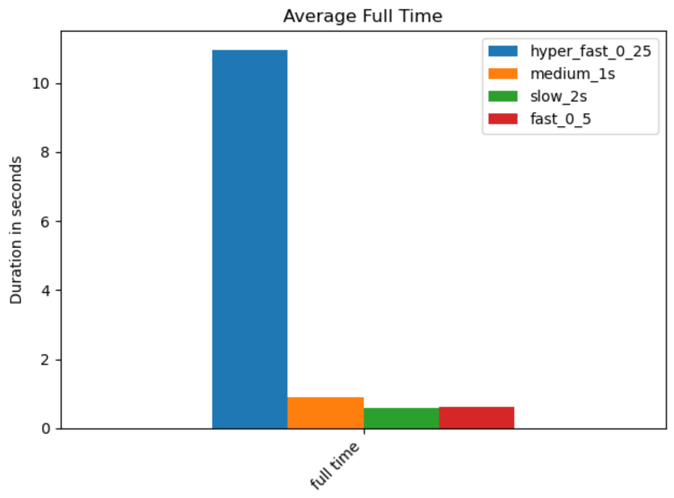


Figure 4: Average time it takes from sending the frame to receiving an alarm

In figure 4 the average latency for each stress level are presented. Unsurprisingly the latency was for all the scenarios where the edge could keep up with the SQS from iot to edge quite similar. For the 0.25 frames per second case the time and amount of messages started to grow linearly. For the 2 second interval we have a average time of 0.575s, for 1 second interval we have 0.885s, for a 0.5s interval we have 0.622s and for 0.25s interval we have 10.953s.

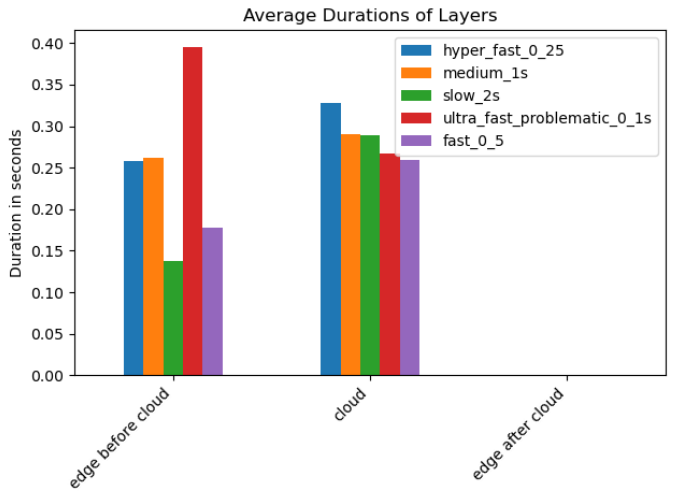


Figure 5: Average time a layer spent on computation

We can see in figure 5 that the cloud layer approximately takes the same time for all stress levels. The edge layer starts to have more computation time as soon as more messages fill the buffer, this might be due to the consumer threads not keeping up with the filtering thread. This effect can be ignored, because in practice we would use asynchronous communication between the edge and cloud layer instead of threads. We only implemented threads to avoid getting our accounts banned. For further improvement of the edge we could use multiprocessing to increase the throughput of one edge instance. In our case, this would just make the edge slower, because the 't2.micro' doesn't have sufficient threads to handle both multiprocessing and multithreading well. The overhead negates the positive gains in this case.

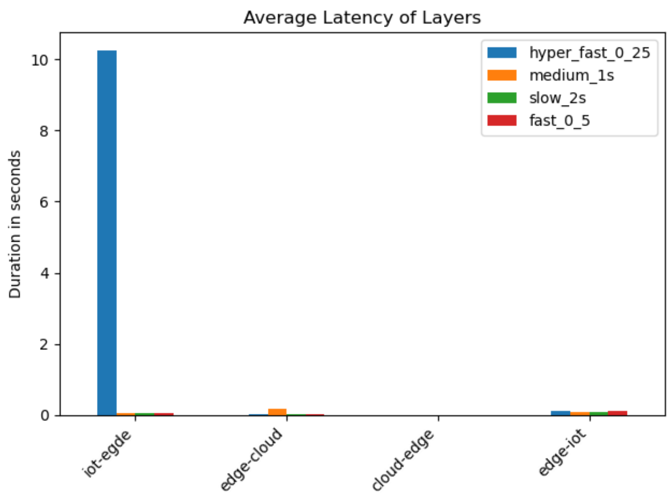


Figure 6: Average latency between layers

In figure 6 the latency from iot to edge starts to get worse, as the messages start to fill up the queue too much, see 0.25 frames/s bar. The edge to cloud latency is nearly the same, because it always calculates a quite similar frame shape/qualitiy. The cloud to edge is not noticeable, because we have a synchronous http connection to the cloud layer for the responses. The time difference from the edge to the iot device is also not that high, because it is like a message forwarding process, with additional sorting out messages and putting additional info to the message.
The figure 6 reveals the weak spot of our system: the communication between iot and edge. As long as the edge layer is able to process the images fast enough, we have zero scalability issues. But as soon as the edge is no longer able to keep up with the iot throughput, the time of processing each message gets longer and longer, theoretically approaching infinity. To prevent this there are more options. First we could improve the performance of the edge as suggested in the

previous part. This would not solve the problem from a theoretical standpoint, because we just increase the maximum throughput of the edge, but practically this can make the difference between a working system or not. Secondly, we could add a max queue size for the sqs queue, removing incoming images when the queue is full. This would have the effect, that the time between iot-edge would no longer approach infinity, but rather a constant. The question is if this approach wouldn't just hide the problem. For example if the number of iot images sent exceeds the computation power of the edge by the factor 5, we would have to remove four-fifths of the total images. With an assumed sending rate of 5 seconds, we would face the issue that there are time sequences of over 30 seconds without any image processed, making our system basically useless. Another advantage of our implementation is that when a edge device fails temporarily, we can recover the unprocessed images when the edge is running again. Lastly we could use the AWS auto scaling feature to make the edge layer bigger. We avoided this, because we assumed that in practice we would not run the edge on AWS but rather a local server.