FACE DETECTION ARCHITECTURE FOR LOW BANDWIDTH SCENARIOS BASED ON DISTRIBUTED COMPUTING AND MACHINE VISION IMPLEMENTED IN KUBERNETES

Author: Carlos Adrián Sánchez Mompó

IIT Advisor: Dr. Jafar Saniie

UPV Advisor: Dr. José Manuel Mossi García

Master's Thesis presented at the Escuela Técnica Superior de Ingenieros de Telecomunicación of the Universitat Politècnica de Valéncia, to obtain the Master's Degree in Telecommunications Engineering (Título de Máster en Ingeniería de Telecomunicación)

Academic year 2019-20

Chicago, August 2020

Acknowledgements

I would like to express my deepest appreciation to Dr. Jafar Saniie, for his great advice and exceptional motivation to pursue the highest level of academic excellence.

My appreciation also extends to David Arnold, for his guidance and care during the development of the project; to all the members of the ECASP research laboratory for their willingness to help; to Dr. José Manuel Mossi for his help and support in the latest stages of the project; and to my colleagues Carlos Mateo and Jose Palermo for being part of this academic journey.

Above all, I want to thank my parents, family and friends, for being there whenever I needed them and for their support; with a special thank you to my father, professor of Civil Engineering at the Technical University of Valencia, for his continuous encouragement and help with the formal revision of the project.

Abstract

In this project, a facial recognition system based on the concepts of machine vision with convolutional neural networks (CNNs), machine learning and distributed computing is proposed, developed and deployed to Azure. The project consists of three parts: the design of an architecture focused on reducing the bandwidth required to transport the images from the cameras to the cloud servers; the design and implementation of the software that compresses the data near the cameras; and the design and implementation of a REST API for image processing easily scalable to large deployments in the cloud.

The reduction of the bandwidth required for the transmission of images from the cameras to the cloud is achieved by pre-processing the data in an intermediate node near their generation through the use of edge computing techniques.

In the pre-processing stage, the faces contained in each of the frames captured by the cameras belonging to that node are identified and extracted. The new images that only contain the faces are then sent to the cloud, where they are classified among the known people. By only sending the portions containing faces instead of sending the entirety of the frames, a significant reduction in the required bandwidth is achieved.

At the cloud, a REST API has been designed and implemented in containers handled by a Kubernetes deployment in the Azure cloud. In these containers, the system that recognizes the faces received from the edge is executed. The load is distributed among the different containers, whose number can be scaled to suit the workload. This API not only allows the recognition of people for whom the model has been trained; it can also incorporate people into the dataset, remove people from the dataset, as well as retrain the recognition model to identify people in the new dataset and start using the new model immediately and without any downtime. All the information extracted in this process is stored in a database that allows its later analysis.

Resumen

En este proyecto se propone y desarrolla un sistema de reconocimiento facial basado en los conceptos de visión máquina con redes neuronales convolucionales (CNNs), aprendizaje máquina y computación distribuida. El proyecto está formado por tres partes: el diseño de una arquitectura centrada en la reducción del ancho de banda necesario para transportar las imágenes desde las cámaras a los servidores en la nube; el diseño e implementación del software que comprime los datos cerca de las cámaras; y el diseño e implementación de una API REST adaptable a grandes despliegues para el procesado de las imágenes en la nube.

La reducción del ancho de banda requerido para la transmisión de las imágenes de las cámaras a la nube se consigue mediante el preprocesado de los datos en un nodo intermedio cerca de su generación mediante el uso de técnicas de computación en el borde (edge computing).

En la etapa de preprocesado se identifican y extraen las caras contenidas en cada uno de los fotogramas capturados por las cámaras pertenecientes a ese nodo. Las imágenes que ya sólo contienen las caras son entonces enviadas a la nube, en la cual se clasifican entre las personas conocidas. Al sólo enviar las porciones que contienen caras en lugar de enviar los fotogramas enteros se consigue una reducción significativa del ancho de banda requerido.

En el extremo cloud, se ha diseñado e implementado una API REST en contenedores manejados por un despliegue de Kubernetes en la nube Azure. En estos contenedores se ejecuta el sistema que reconoce las caras recibidas desde el edge. La carga está distribuida entre los diferentes contenedores, cuyo número puede variarse para adaptarse a la carga de trabajo. En esta API no sólo se permite el reconocimiento de personas para las que el modelo se ha entrenado; también se puede incorporar personas al set de datos, eliminar personas del set de datos, así como reentrenar el modelo de reconocimiento para identificar las personas en el nuevo set de datos y empezar a usar el nuevo modelo de forma inmediata y sin downtime. Toda la información extraída en este proceso es almacenada en una base de datos que permite su posterior análisis.

Resum

En este projecte es proposa i desenvolupa un sistema de reconeiximent facial basat en els conceptes de visió màquina en xarxes neuronals convolucionals (CNNs), aprenentatge màquina i computació distribuïda. El projecte està format per tres parts: el disseny d'una arquitectura centrada en la reducció de l'ample de banda necessari per transportar les imatges des de les càmeres als servidors en el núvol; el disseny i implementació del software que comprimix les dades prop de les càmeres; i el disseny i implementació d'una API REST adaptable a grans desplegaments per al processat de les imatges en el núvol.

La reducció de l'ample de banda requerit per a la transmissió de les imatges de les càmeres al núvol s'aconsegueix per mig del preprocessat de les dades en un nodo intermig prop de la seua generació per mig del ús de tècniques de computació a la vora (edge computing).

En l'etapa de preprocessat s'identifiquen i s'extreuen les cares contingudes en cada un dels fotogrames capturats per les càmeres pertanyents a este node. Les imatges que ja només contenen les cares són llavors enviades al núvol, en la qual es classifiquen entre les persones conegudes. A l'només enviar les porcions que contenen cares en lloc d'enviar els fotogrames sancers s'aconsegueix una reducció significativa de l'ample de banda requerit.

A l'extrem cloud, s'ha dissenyat i implementat una API REST en contenidors manejats per un desplegament de Kubernetes en el núvol Azure. En estos contenidors s'executa el sistema que reconeix les cares rebudes des del edge. La càrrega està distribuïda entre els diferents contenidors, el nombre pot variar-se per adaptar-se a la càrrega de treball. En esta API no només es permet el reconeiximent de persones per a les que el model s'ha entrenat; també es pot incorporar persones a el set de dades, eliminar persones de el set de dades, així com reentrenar el model de reconeiximent per identificar les persones en el nou set de dades i començar a utilitzar el nou model de forma immediata i sense indisponibilitat. Tota la informació extreta en este procés és emmagatzemada en una base de dades que permet la seua posterior anàlisi.

TABLE OF CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

Table of Acronyms

ACRONYM	DEFINITION
1080P	1920x1080 Progressive (Video)
5G	Fifth generation technology standard for cellular networks
AI	Artificial Intelligence
AKS	Azure Kubernetes Service
API	Aplication Program Interface
APP	APPlication
BIPA	Biometric Information Privacy Act
BSD	Berkeley Software Distribution
CA	Certification Authority
CLI	Command Line Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DB	DataBase
DDOS	Distributed Denial of Service
DEVOPS	Development Operations
DNA	DeoxyriboNucleic Acid
DNN	Deep Neural Network
ESXI	Elastic Sky X integrated
EU	European Union
FPS	Frames Per Second
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
H264/AVC	Advanced Video Coding
H265/HEVC	High Efficiency Video Coding
HP	Hewlet Packard
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ID	IDentifier
IOT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
LFW	Labeled Faces in the Wild
LTS	Long Term Service
MTCNN	Multi-Task Cascaded Convolutional Neural Networks
NFS	Network File Storage
ONNX	Open Neural Network eXchange
OPENCV	Open source Computer Vision library
OS	Operative System
PHP	PHP: Hypertext Preprocessor
RAM	Random Access Memory
RESNET	Residual Network
RESTFUL	REpresentational State Transfer
RESTPLUS	REpresentational State Transfer Plus
SP	Service Provider
SQL	Structured Query Language
SSH	Secure SHell

SVM	Support Vector Machine
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TPU	Trensor Processing Unit
UDP	User Datagram Protocol
URL	Uniform Resource Locator
US	United States
UWSGI	(u)Web Server Gateway Interface
VCPU	Virtual Central Processing Unit
VM	Virtual Machine
VPN	Virtual Private Network
YAML	YAML Ain't Markup Language

Chapter 1: INTRODUCTION

At the beginning of the computing age, the data that were obtained by the different systems were processed locally. This was in part due to the lack of a telecommunications infrastructure capable of sending these data across cities.

Nowadays, in the cloud age, almost all the data processing has been moved to “the cloud”, i.e. the data are being sent from where they are generated to remote servers for processing. However, due to the popularization of the Internet of Things (IoT), there has been an increase in the generation of highly-distributed data, which leads to doubling the number of IP connections every 2.5 years [1], which poses a challenge for the underlying data communication networks.

The cost of data transport in some high-bandwidth applications, such as video surveillance and analysis, is becoming a barrier for the advancement of the technology, as these distributed data can quickly saturate the traditional pyramidal structure of Internet Service Providers’ (SP) networks.

It is because of this that both the bandwidth required and the distance that the data need to travel for an application must be reduced as much as possible. With this goal in mind, in these high-bandwidth applications, the data needs to be compressed or pre-processed near the devices that are producing it.

The tendency of bringing data processing nearer to the devices that are generating them is known as edge computing, and it is predicted to be one of the technological trends of the decade [2], as it becomes the most cost-effective and highest performance solution in which many new technologies, such as 5G, rely heavily.

1.1 MARKET OVERVIEW

In the following subsections, an overview of the video surveillance market is provided. This overview focuses on both the home and large-scale markets, identifying the main actors and providing a brief analysis of their products.

1.1.1 Home and Small Business Security

In the home and small business market, the customers have a low count of cameras on-premises, and it is common to have most of the work performed in the cloud.

1.1.1.1 Ring

The Ring ecosystem [3] provides solutions and devices for home automation and home security. The device catalog includes a large number of devices which integrate a camera for monitoring purposes.

The processing on the devices themselves is limited in most cases to motion detection. Whenever motion is detected, the camera performs a predefined function. This function could be to record the images and upload them to the cloud or to send a message to a device through the cloud.

The images are stored in the cloud servers, where not much processing is performed. The storage of these images is part of a service provided by Ring under the name of Ring Protect.

As the system does all the processing in the camera in order to save bandwidth, the processing is limited and it may not provide the user with as much insightful information as offered by other systems.

1.1.1.2 Nest

The Nest ecosystem [4] provides mainly home automation services, although it also has a range of devices that target home security. Their home security catalog includes cameras, alarm systems, locks, and other devices.

The processing is carried out mainly in the cloud, since the cameras only performs motion detection.

The system processes in the cloud the images sent by the different cameras. Via this processing, it decides whether there is a relevant event. It then classifies all events in a series of categories using artificial intelligence. This service is called Nest aware.

The problem with a service that relies so heavily on the cloud computation, and that provides services with 24/7 cloud recording is that each camera requires an expensive fixed upload bandwidth (approx. 1.2 Mbps/camera according to Nest, but usually 5 Mbps/camera for a typical H264 1080P camera). However, in the type of scenario that this system is used (where there are few cameras per location), this may be the best solution, as high-level processing in the camera is computationally expensive and performing local processing is expensive as well, as a new computation device is required.

Furthermore, by moving all the processing to the cloud, the system can provide the user with much more useful information than it would be able to provide if all the processing were to be done in the cameras themselves. However, for large deployments, this system would prove inefficient, due to the bandwidth requirements previously mentioned.

1.1.1.3 Xiaomi

The last Home automation system in this brief analysis is the Xiaomi Smart Home ecosystem [5], which is composed by a great variety of sensors, actuators, cameras, alarms, etc. Regarding the security cameras, there are several models that offer different on-device processing capabilities.

The processing is mostly performed in the cameras, where the images are analyzed to determine whether there is motion in the scene, and if so, the camera starts recording locally.

There are also some cameras in the catalog that perform AI humanoid detection to increase the versatility of the motion detection system.

In this system, both the processing and the storing of the video is performed locally in the cameras. It is because of this that the cloud serves as a mere aggregation and user access platform.

This type of system does not saturate the communications networks when deployed in large scale as others do. However, the system requires much more local processing in the cameras themselves, which may not be the most efficient way to process things. Also, as the end devices lack the processing capabilities of the cloud, the information provided to the user is very limited when compared to other systems that process the information in the cloud.

1.1.2 Lage-Scale Security

In the large-scale business market, the customers have a high number of cameras on-premises, and it is common to have most of the work performed on-site.

1.1.2.1 Anyvision

The Israeli face recognition company installs high-performance and large-scale video surveillance systems for companies and governments all around the world. These systems can be configured in many ways, although based on some recent installations in Spain, the system seems to be installed near the cameras if there is a great mass of cameras in a given location. This means that the camera does no processing, so any IP video camera may be used with the system.

It also means that there are servers installed on-premises to process the video generated by the cameras. This is done by processing the images and identifying the individuals and their actions. The processed data can then be transmitted to a central controller. In the central controller the data are aggregated and a general view of the system is offered with real-time analytics.

This type of installation poses a problem: either you have the cameras transmitting the video over the internet or you process it on-site. If the video is processed on-site, the installations need to be modified in order to accommodate the system's servers.

A better solution than the type of installation being reviewed could be to only partially process the images on-site just enough to reduce the bandwidth without requiring the installation of a server room on-site. This could be something that is already done by Anyvision in some deployments, although, as their technology is closed source, it is difficult to tell without more information.

1.2 Open Source Projects

There are many open source projects that target image processing and face recognition, some of which use machine learning models to achieve this. In the machine learning field, the biggest companies (Google, OpenAI, etc.) publish most of their work allowing access to the international scientific community.

In this section, an overview is given of some of the publicly available machine learning-based face recognition projects, as a general view of the state of the art of this rapid-evolving field.

1.2.1 OpenFace

The OpenFace project [6] is an implementation of the FaceNet face recognition architecture proposed by Google researchers [7]. The OpenFace architecture, as shown in Figure 1, implements image detection, image transformation (to get the face straight), image cropping, feature embedding generation, and feature embedding classification.

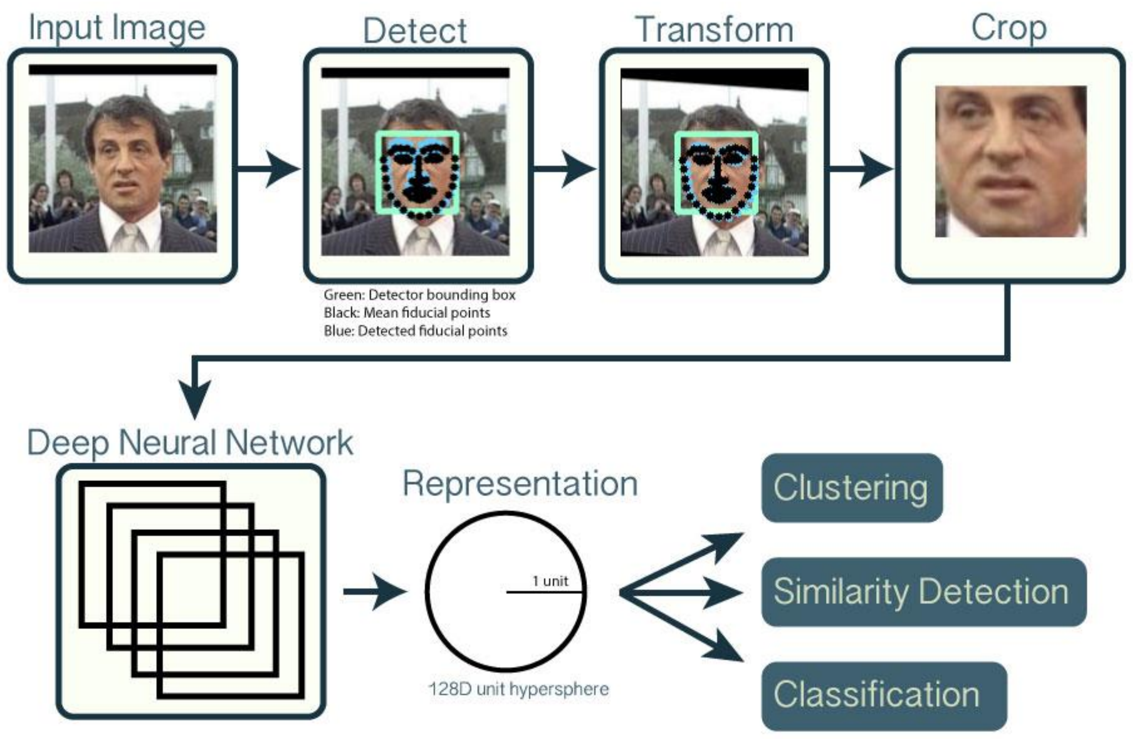


Figure 1: Diagram of the OpenFace Architecture

The architecture on which OpenFace is based is commonly found in many other implementations. Some of these implementations even share some of the Python code of the OpenFace implementation. This is probably due to OpenFace being one of the first open source projects to use this face recognition architecture.

1.2.2 A FaceNet Implementation

There is a very popular implementation of the Google FaceNet architecture, which also goes by the name of FaceNet [8]. It uses the Tensorflow platform [9] to manage the DNN model required to obtain the embeddings and to run the Multi-Task Cascaded Convolutional Neural Networks (MTCNN) face detection.

As it is based on the FaceNet architecture, the system proposed in this implementation is quite similar to that of the OpenFace project, going so far as to share some of its base code with it.

This project not only provides the architecture and the code to run the face recognition (in Python), but also two pre-trained face embeddings generation DNNs that can be used in conjunction with a face detector and a classifier to perform the face recognition.

This system and its pre-trained face features embedding extractors can be commonly found in other open source projects. This is due to the complexity and especially the cost of training a modern DNN from scratch to perform embedding generation.

1.2.3 A FaceNet Docker Implementation

This final project [10] is also based on the FaceNet architecture by Google, and on the FaceNet implementation [8] previously discussed. It provides another Python implementation of the base algorithm which is also prepared to run inside a Docker container.

It is somewhat different from the other implementations as it uses Pytorch FaceNet [11] as the basis for face detection and feature extraction. The Pytorch FaceNet provides pre-trained models for face detection and recognition. These models are the same as the ones offered in [8], showing how common the use of these pre-trained models is in the open source community.

This project has served as the inspiration and basis for the present work, as it has provided a starting platform which has then been heavily modified and adapted to the specific needs of this work. The modification has mainly consisted in adding many functions, modifying the existent functions to add much more functionality, changing the classifier and other libraries used in the original implementation, and improving the efficiency of the training process; and all of this, using an edge (distributed) computing approach in the architecture design.

Chapter 2: PROPOSED SOLUTION AND DESIGN

2.1 THE SOLUTION – OBJECTIVES

The objective of this project is to design a system that obtains useful information from a video recording system while requiring a small amount of data bandwidth and a low processing power in the edge computing. All of this while providing a smart scalable solution with high availability.

To achieve these goals the system architecture designed and implemented in this document divides the image processing into two parts: edge computing and cloud computing.

2.2 THE ARCHITECTURE

The face recognition architecture implemented in this project has been designed to perform most of the processing in the cloud, as many of the systems seen in the introduction. However, it has a key distinction: to reduce the bandwidth required between the cameras and the cloud, the data are pre-processed in edge computing servers close to the cameras. This enables to reduce the amount of data transmitted from the edge to the cloud. The bandwidth reduction is achieved by extracting face images candidates in the edge, and sending them to the cloud for final extraction and recognition, instead of sending the whole video streams.

The edge face extraction is optimized for speed, while the cloud face extraction is optimized for accuracy. This way, as explained in this section, the bandwidth compression is achieved without a great computational cost in the edge, enabling the implementation of the system in small and compact edge nodes, as further detailed in following sections. The edge nodes perform a preliminary selection of candidate faces, while the cloud nodes extract the real faces and discard the non-face images.

In the distributed computing scheme shown in Figure 2, the size of the data transmitted is reduced and the useful information is increased in the path from the cameras to the database. In other words: the connection between the cameras and edge computing devices has the greatest bandwidth requirement while the connection between the edge computing devices and the cloud has a lower bandwidth requirement as the information is more compact. Also, the information sent from the cloud processing to the database and NFS storage is even lower in size, while containing all the relevant information.

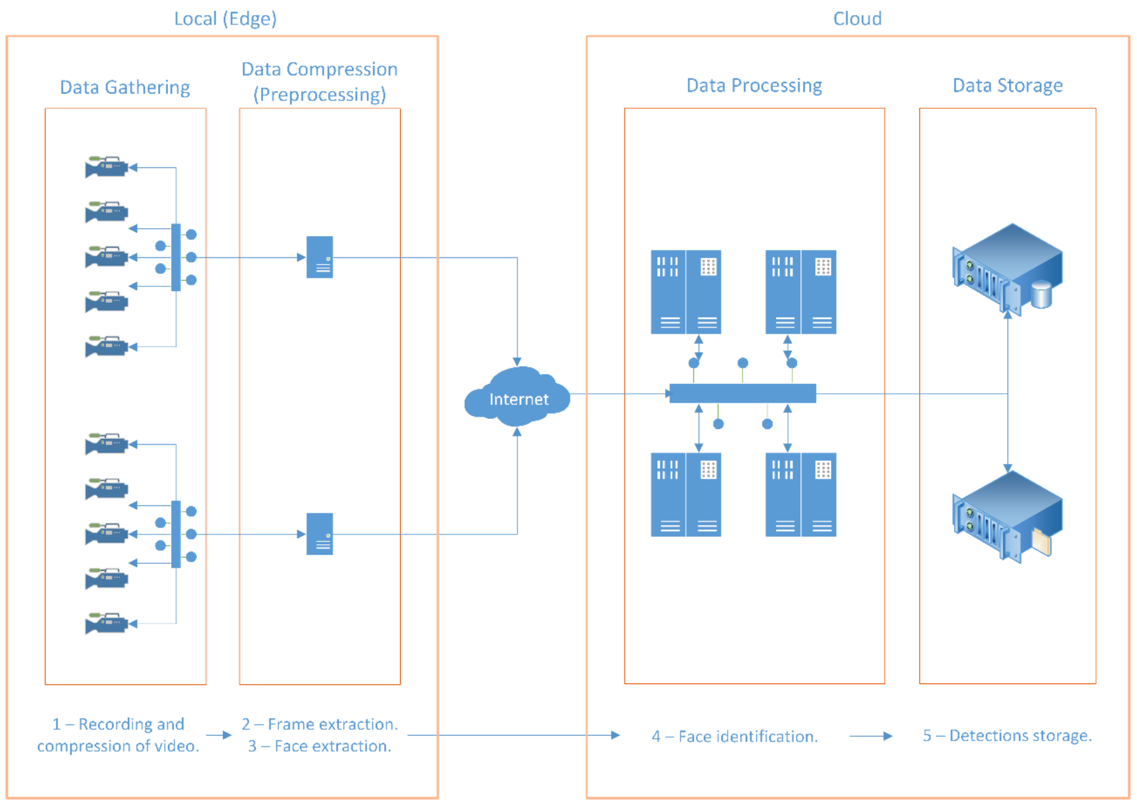


Figure 2: Architecture of the Distributed Computing

The diagram found in Figure 3 represents the same architectural scope as the one in Figure 2, although in this diagram, the focus is on a single data path, showing both the logical and physical connections of the system. As may be seen, the video streams generated by the cameras are fed into the edge computing module, where the faces are extracted. Note that the faces will be processed and extracted again in the cloud as the algorithm used in the edge processing is optimized for speed and the algorithm used in the cloud for accuracy.

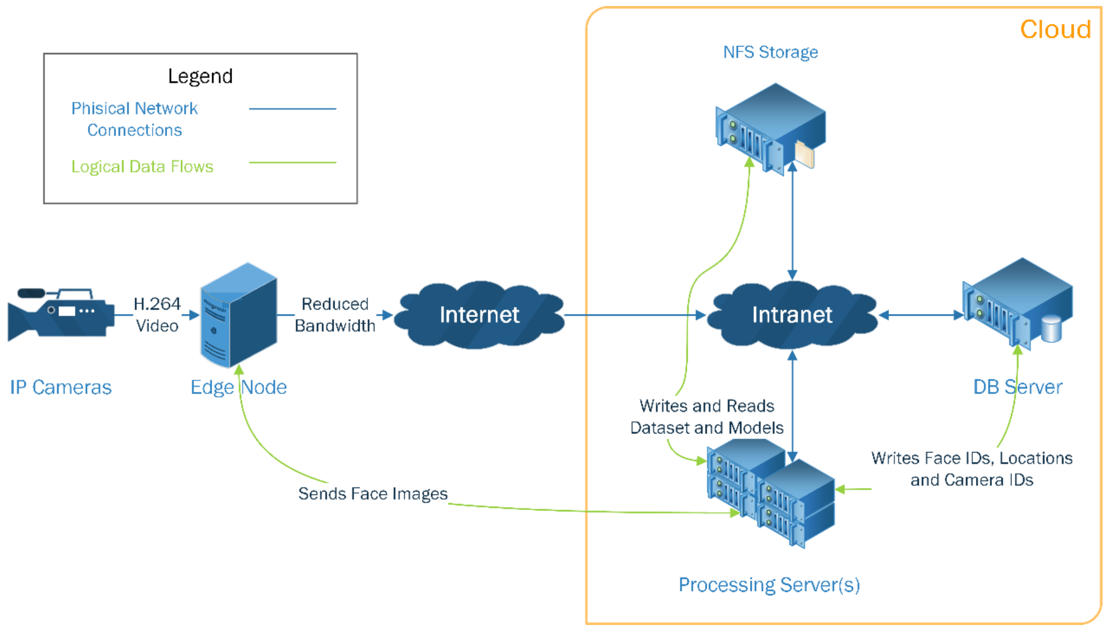


Figure 3: Network and Data Flow of the Distributed Computing Architecture for one Camera Stream

The edge node decodes the original high-resolution video, extracting the frames. These frames are processed looking for face candidates. The algorithm used to detect the faces in the frames is optimized for speed, not for accuracy, as the edge processing power is assumed to be low.

To avoid missing faces, the minimum confidence level to determine that a face candidate is sent to the cloud is set to a low enough value. Therefore, as seen in Figure 4, there will be a certain percentage of face candidates that do not really contain a face in the image. This is not a problem, as the images are then sent to the cloud to be processed again, but this time with a better face detection algorithm that will discard the incorrect images.

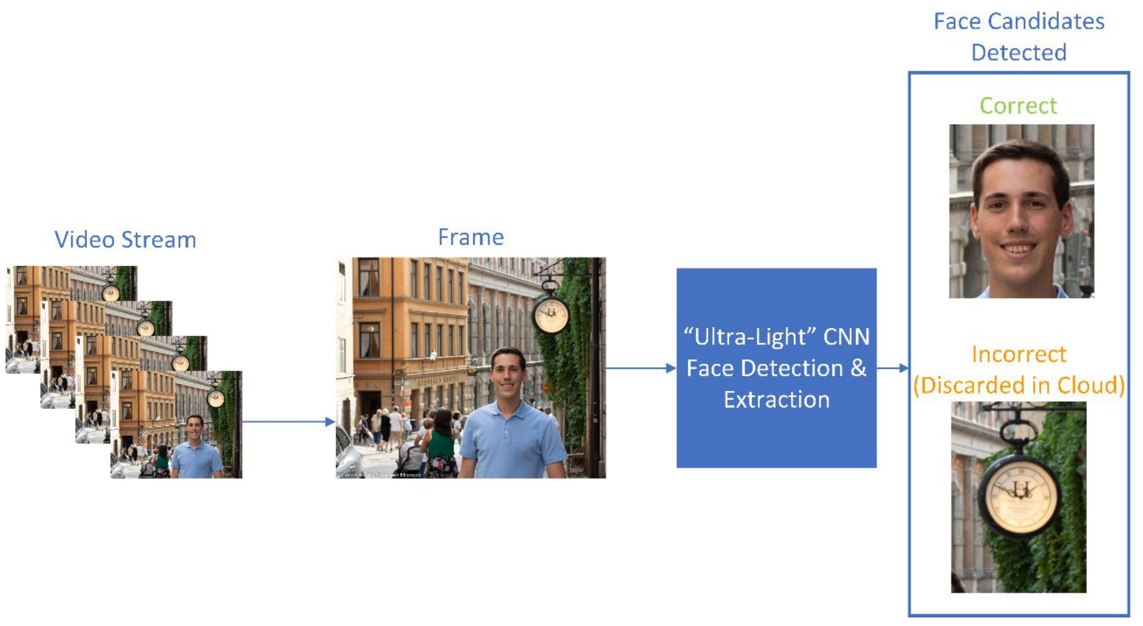

Face Candidates Detected
Correct
Video Stream
Frame
"Ultra-Light" CNN Face Detection & Extraction
Incorrect (Discarded in Cloud)

Figure 4: Face Candidates Extraction on Edge Computing

The images received by the Cloud API are then analyzed by one of the most accurate face detectors available, as there is no processing power limitation: the MTCNN face detector [12]. This is done, as shown in Figure 5, to discard the images that do not contain a face and to prepare those that do contain a face for the feature extractor.

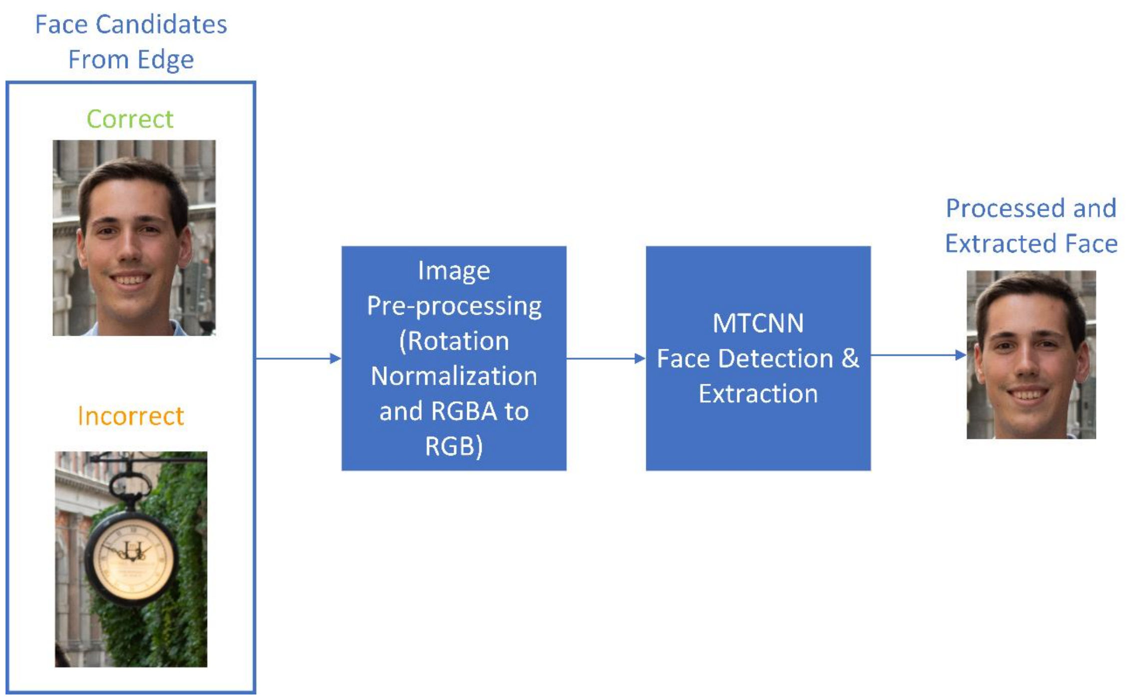

Face Candidates From Edge
Correct
Incorrect
Image Pre-processing (Rotation Normalization and RGBA to RGB)
MTCNN Face Detection & Extraction
Processed and Extracted Face

Figure 5: Image Pre-processing and Extraction as Input for the FaceNet-like Architecture

The faces detected and processed are then fed to the face recognition and classification system shown in Figure 6, which is based on the Google FaceNet [7] face recognition architecture. This system uses a Convolutional Neural Network (CNN) trained to extract feature vectors (embeddings) [13] connected to a multiclass Support Vector Machine [14] (SVM) classifier. This classifier is trained to classify an embedding extracted from an image of one of the people included in the training dataset.

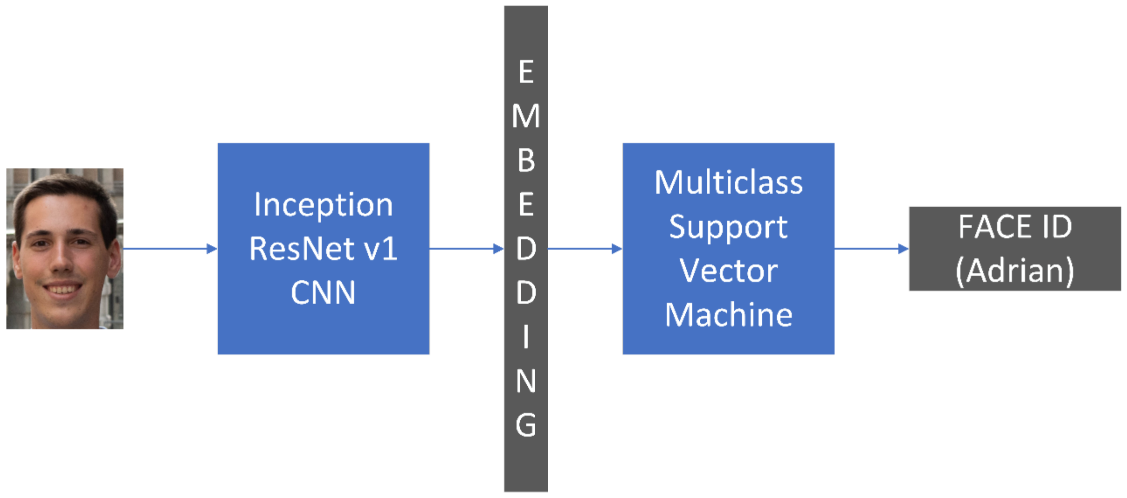


Figure 6: Google FaceNet Face Recognition Architecture with multiclass SVM

Note that apart from the candidate face images, the camera ID and location in which the image was captured are also sent by the Edge node to the Cloud processing servers; these data are stored along with the face ID of the face image into the Detections table inside the DB server. This can be used to trace the movements of the people in the locations being monitored.

For greater flexibility and to allow deployment sizes that range from less than 10 cameras to 1000s of cameras, the data processing stage has been designed so that it can be scaled in an automated fashion by the use of cloud infrastructure providers.

As shown in Figure 7, a Kubernetes cluster is used to process the images and both an NFS server and a DB server are used as common storage for all the instances that are running in the Kubernetes deployment in the different nodes of the cluster.

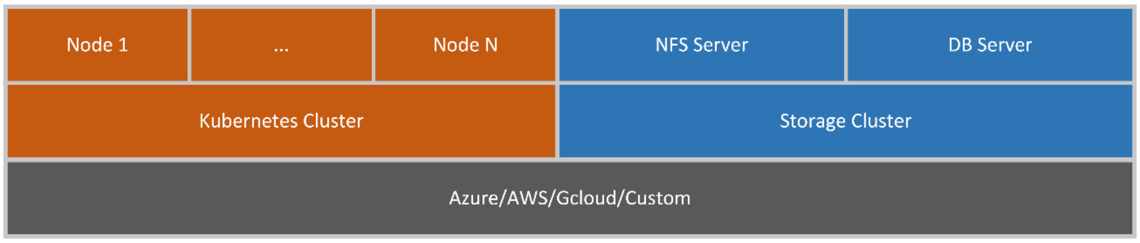


Figure 7: Kubernetes and Storage Architecture

All the data required to train and to run the facial detection system and all of the accompanying functions that are discussed in later sections of this document are stored in the NFS server and DB server, (see Figure 7).

This processing architecture allows for self-healing and implements on-the-fly updates to both the classification model (when there is a change in the people that the system is trained to recognize) and the API backend code, without any downtime. This means that it is not only a proof of concept or a laboratory test, but the start of what could be a facial recognition architecture that with a few modifications could be deployed to production.

With this method, state-of-the-art tracking, recognition and behavioral analysis methods can be scaled up to be implemented in commercial, governmental or military applications, even in areas where it is not feasible neither to have the processing power required to process the images on-site nor to have a good connectivity to the central processing servers to send the unprocessed video feeds.

2.3 INTEGRATION WITH MOBILE ANDROID AND IOS APP API

Two projects have been developed simultaneously to create a smart distributed computing and low bandwidth surveillance system: a mobile APP for Android and iOS [15] and this project.

To allow for the interconnection between both projects, an API call has been implemented (face-recognition/send-results) that sends the detection data to the cloud infrastructure where the Mobile APP API runs. This allows for easy interconnection between both projects.

Therefore, the connectivity scheme for the operation of both systems together involves, as shown in Figure 8, the edge computing device, the Cloud API and the APP API.

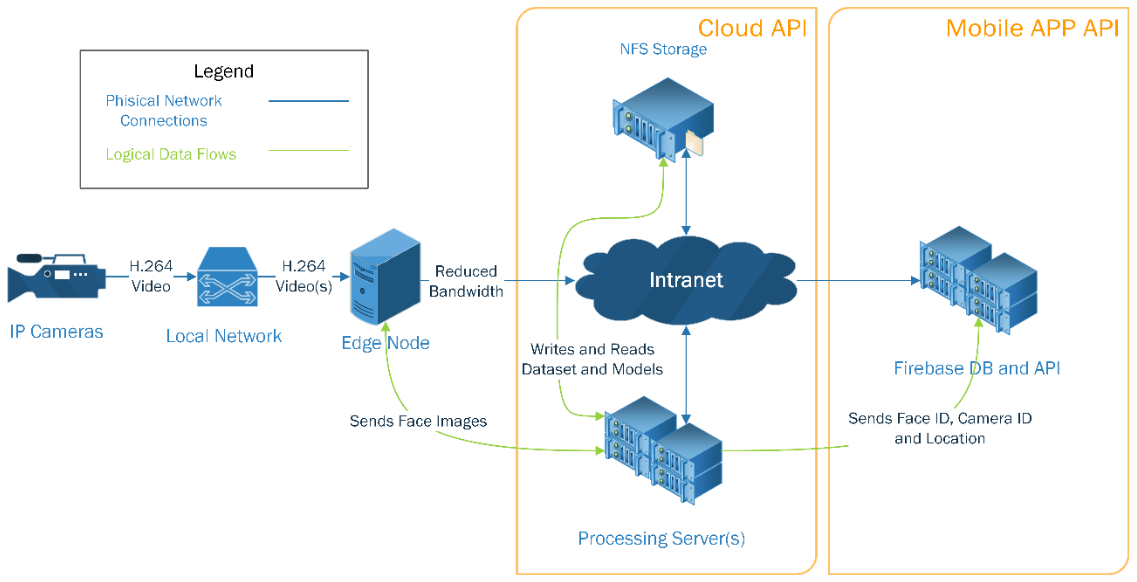


Figure 8: General Diagram of Interconnection with the Mobile APP API

When joining both projects, this system gains an interface to interact with the user of the system providing statistics regarding detections and locations. The goal is to demonstrate how the system could be used in a security surveillance scenario.

2.4 THE ALGORITHM

As seen in the previous subsections, the FaceNet-like face matching algorithm used in this project has two steps: face features extraction and face matching. In the first step, the different features conveyed in the facial image are used to generate an embedding vector that defines the face. In the second step, the vector is used to pair the face image with the best matching face identity from the ones in the Dataset (providing a confidence level for the match).

2.4.1 Feature Extraction

For the face feature extraction, a Convolutional Neural Network (CNN) is used. This type of network is characterized by containing convolutional layers, fully connected layers, and many other supporting layers such as pooling, dropout, etc.

The convolutional layers are steps of the network that perform a correlation operation (not a proper convolution, although the name could seem to imply it) of the input of the layer with a filter. This correlation is stored in the output of the layer. Usually, in this type of layers the dimensions of the image are reduced while the depth or feature information is enlarged in each of the steps.

The pooling layers are used in between convolutional layers to pool or reduce the dimensions of the matrix, usually reducing the spatial resolution while keeping the feature depth. This way, the characteristics found in the image are made less specific to the location of the characteristic and more specific to the type of characteristic.

The fully connected layers consist of a single layer of neurons in which each one of the neurons is connected to all of the outputs from the previous layer. This is the reason behind the “fully connected” name, as each one of the neurons of a particular layer are fully connected to the previous layer.

In the CNN, the image analysis usually starts with a number of convolutional layers, pooling layers, and other supporting layers, and it ends with a series of fully connected layers. The convolutional layers extract the features from the input image and the fully connected layers use these features to determine what should be placed at the output of the CNN.

In this project, the output of the network needs to be an embedding that can be used as an identifier of the person in the image. This is achieved by training the network to obtain embeddings that are as similar as possible for images that belong to the same person, and as different as possible for images that belong to different persons.

2.4.2 Embedding Classification

Once the face embedding has been obtained, the group to which it pertains, (i.e. the identity of the person in the image, from which the embedding has been derived) needs to be determined. This can be achieved by many different methods, most of which require to have a database in which the embeddings associated to the different identities used for training purposes are stored.

A simple method to determine which of the known identities best matches the embedding is to find the closest known embedding, i.e. the embedding that has the lowest distance (Euclidean, Manhattan [16]) to the current embedding in the database.

A better method would be to obtain the average of the embeddings for a given identity and then find the average embedding with the lowest distance to the embedding obtained by the Feature Extractor for the current image.

A refinement of the first method is to use more than one embedding in the comparison: instead of finding the closest embedding we could find the K-closest embeddings, and set the identity as the one that appears most times in the K-nearest embeddings. This is known as the K-nearest neighbors method [17].

Another very popular method that may achieve even better performance than the previous methods is the Support Vector Machine (SVM) [14]. With standard SVM, a model is trained to differentiate between two classes or options in an optimal manner. This can be expanded for more than two classes. Because of its accuracy advantages, this is the method that will be used in this project; specifically, a linear kernel variant of the multiclass SVM.

2.5 Feature Extractors

The state of the art in Convolutional Neural Networks is analyzed in this section to provide a broad vision of the different feature extraction architectures and why the pre-trained Inception ResNet v1 [13] is used in this project to obtain the embedding for the input image.

There is not a one-fits-all feature extraction architecture. The best selection depends on the intended use. In this project, the feature extractor must perform a low number of computations per iteration, as there may be many faces being processed per minute, reaching thousands depending on the deployment size. Simultaneously, it is required to maintain a high accuracy, as the face vector embeddings must separate the different identities with enough confidence for the intended number of users.

There is a great number of feature extraction architectures that are being studied and proposed by the scientific community. The two most recent and innovative methods that could be used for this project are analyzed in the following subsections, as well as an additional third version which, while not being the most efficient for the task, is the best available in a pre-trained form. The last one fits very well the needs of this project, as its focus is not to train a new type of CNN for feature extraction (which would be a project in itself), but to design, implement and test a new video surveillance and face recognition architecture.

2.5.1 MobileNet v3

The MobileNet [18] CNN is specifically designed for its use in mobile devices. It is adapted for the execution in mobile CPUs using "hardware aware network architecture search" techniques. It achieves very high accuracy in the ImageNet [19] dataset with a very low per-frame latency as shown in Figure 9 for different parameter sizes, enabling for real-time processing even in mobile CPUs.

The fact that it is designed to use a limited amount of computing power in CPUs while maintaining a high accuracy suits the purpose of this design, and enables the architecture to process the face images in any system, independently of the type of processing devices installed (without requiring specialized hardware such as GPUs or TPUs) and with a low memory usage.

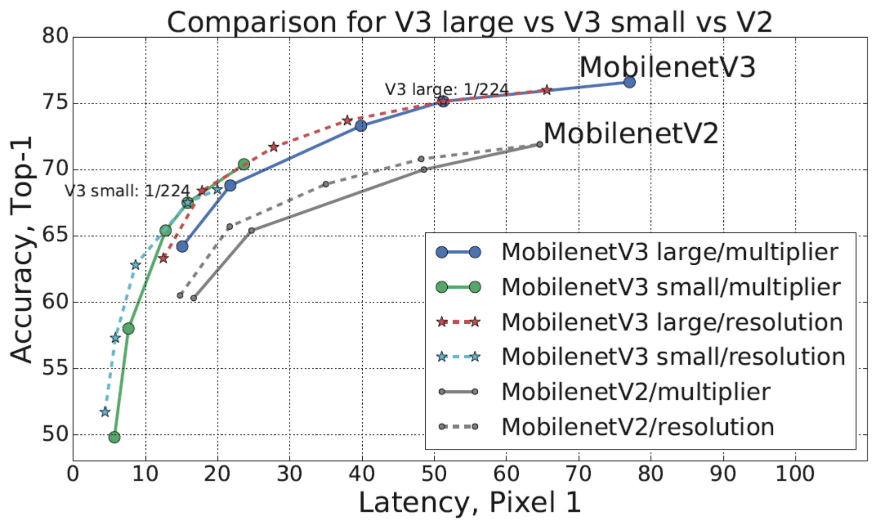


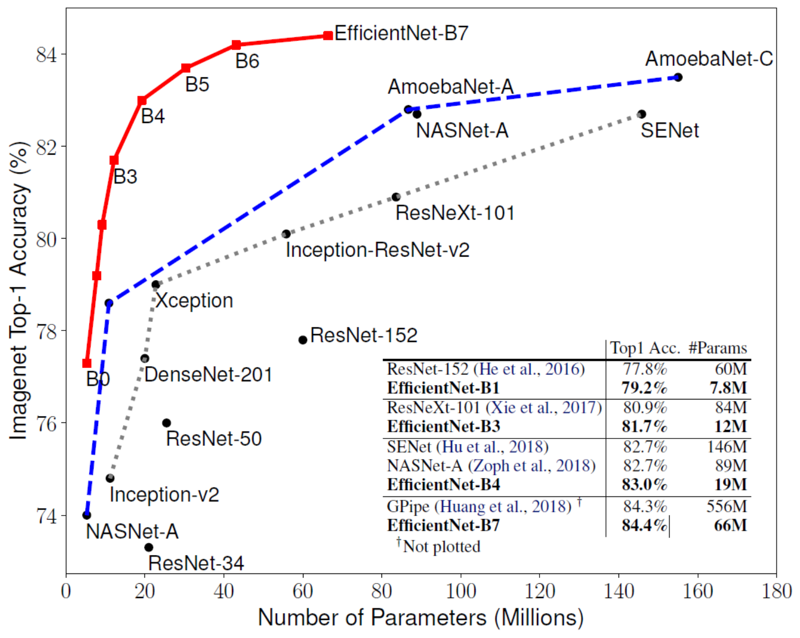
Figure 9: Performance of MobileNetV3 as a function of different multipliers and resolutions in the ImageNet Top-1[1] [18]

All these characteristics make it a good option for applications that require a high-throughput, and good accuracy.

2.5.2 EfficientNet

What started as a project to find a better way for scaling up and down CNN architectures, ended producing a new CNN model known as EfficientNet [20]. These Convolutional Neural Networks are based on an architecture of their own and use the same scaling method proposed by the researchers to create a series of CNNs that perform better than comparable architectures at ImageNet and other tests (see Figure 10). They achieve better performance for the same number of CNN computation parameters, as well as less parameters and calculations for the same target accuracy. This efficiency is exactly what is required by projects like this to achieve the desired accuracy and throughput without the use of a large and expensive resource pool.

[1] Top-1 accuracy: the accuracy of the neural network when checking if the ground truth signal matches the class predicted with the highest probability. As opposed to Top-5 accuracy, where the comparison includes the top 5 classes with the highest predicted probability.



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.2%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.7%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**83.0%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

†Not plotted

Figure 10: Performance of EfficientNet as a function of the number of parameters. Comparison with other network types in the ImageNet Top-1 [20]

When compared to the MobileNet architecture (which is an architecture already designed for low parameter count), the variant B0 of EfficientNet obtains a better accuracy with a slightly lower number of parameters, as shown in Table 1. However, the number of parameters is not the only consideration when choosing a network, as the number of operations required to compute an iteration, while closely related to the number of parameters, is not equivalent.

Network	Top-1 Accuracy (%)	Parameters (Millions)
EfficientNet B3	81.7	12
EfficientNet B1	79.2	7.8
EfficientNet B0	77.3	5.3
MobileNet V3-Large	75.2	5.4
MobileNet V3-Small	67.4	2.5

Table 1: Accuracy vs. number of parameters for EfficientNet and MobileNet V3

Given the good efficiency and high adaptability of the EfficientNet feature extraction CNN, it is a candidate for future improvements of the facial recognition architecture designed in this project.

2.5.3 Inception ResNet v1

Finally, the CNN that has been used for this project is the Inception ResNet v1 [21]. This CNN is based on the Inception v3 and Inception v4 architectures, to which it adds Residual Networks (ResNet).

The residual networks consist on skip connections that jump over some of the layers of the CNN. This method keeps the long processing path that can extract deep characteristics and ads a shorter path that allows the CNN to be trained more efficiency by avoiding some problems such as the vanishing gradients.

The performance of this type of network is said to be between those of Inception v3 and Inception v4, although the performance-per-computation or performance-per-memory required metrics would vary depending on the type of data that the network has to learn to predict, as well as the computing capabilities of the hardware on which it is trained and run.

One of the most relevant reasons why Inception ResNet v1 is the CNN used in the feature extraction stage of this project is because there is an open-source, well-known, pre-trained version available for face embedding generation [13]. This version has been trained over roughly 100,000 iterations, which requires both a great computation power and a long time.

2.6 Proposed Edge Computing Devices

Although in this project the edge computing is not implemented in an edge device, candidates for this task are selected in this section for possible practical implementations of the system.

The edge nodes perform video and image manipulation and compression operations. These types of operations require the use of a GPU to be performed efficiently. The goal when selecting a device is to reduce the cost, the energy consumption and the infrastructure required for its operation, while increasing the throughput and number of camera stream that it can process simultaneously.

Given the target characteristics of the devices, the best choice is a single board computer. As the GPU is an important requirement for video handling, the TPU and CPU-only options are discarded. The best solutions seem to be provided by NVIDIA Corporation [22].

The devices considered are part of the NVIDIA Jetson family [23]. Shown in Table 2 are some of the most relevant characteristics for the proposed usage, such as the decoding and encoding throughputs, measured in video streams per second.

Device	**Encoding (HEVC** [24]**)**	**Decoding (HEVC)**
Jetson Nano	Up to 4x1080p@30 fps	Up to 8x1080p@30 fps
Jetson Xavier NX	Up to 12x1080p@30 fps	Up to 32x1080p@30 fps
Jetson Xavier AGX	Up to 32x1080p@30 fps	Up to 52x1080p@30 fps

Table 2: Encoding and Decoding capabilities of the different devices of the Jetson family

According to Table 2, the Jetson Nano could theoretically be used for installations of up to 4 high-resolution cameras, while the Jetson AGX Xavier module could be used for installations of up to 32 high-resolution cameras if only the decoding and encoding capabilities were to be considered.

However, there are more things to be done by the edge processing apart from video decoding, such as face detection. The face detection performed by the edge device would probably be implemented in the GPU, as these devices are very efficient at processing parallel data such as images or CNN inference. Performing the decoding, resizing, face detection, cropping and compression operations would probably reduce the number of streams by a factor of 2, and the frame rate to 10 fps for an efficient algorithm, leading to a reduced performance.

In the case of the Xavier NX and the Xavier AGX, thanks to the Neural Accelerators included in these devices, the face detection algorithm could be accelerated, and the GPU could be offloaded to be able to process more streams per device. The downside is that these devices entitle a cost much higher than the more basic Jetson Nano.

2.7 Cloud Computing

The Cloud API was planned to be implemented in a Cloud provider, and after reviewing the most prominent available choices (AWS [25], GCloud [26] and Azure [27]), Microsoft Azure cloud was chosen for its growing success and mature Kubernetes support.

Prior to the implementation in the Azure cloud provider, the system was installed in an HP 360e (16 c / 32 t, 96 Gb RAM) server with VMware's ESXi Hypervisor [28]. However, as the architecture had been planned to use devices available in the cloud such as load balancers, it was later migrated to the Azure cloud and adapted to the specific requirements of this cloud provider's Kubernetes implementation.

The Docker images used in the Kubernetes deployment were also built/compiled in an online service (Docker Hub [29]) to have them readily available for downloading by the Cloud infrastructure. This system was connected to a GitHub [30] repository to which all the changes were committed, and from which automated builds were issued.

Chapter 3: SCHEDULE AND TASKS

3.1 SCHEDULE OF THE PROJECT

The project definition and initial research started in the middle of the Fall semester of 2019, and its development and implementation continued throughout the Spring and Summer semesters of 2020.

The finalization date of the project is around mid-August. The last month has been dedicated to both conclude the project by improving some routines and styles, and to write the project final report.

3.2 TASKS PERFORMED

The research and development conducted during the realization of this project could be divided in the following tasks:

- Setting of project objectives
- Project outline
- Research of the state of the art
- Complete project definition
- Testing several codes and bases for the face recognition
- Final design of the architecture
- Coding of the image processing
- Coding of the Cloud API
- Implementation of the Cloud API in Docker
- Implementation of the Cloud API in Kubernetes
- Implementation of the Kubernetes cluster in Azure
- Testing and improvement of the system
- Write interfacing functions to test the system
- Add database and database functions to the Cloud API
- Write edge programs to interact with the Cloud API
- Create the final demo functions to interact with the Cloud API
- Write the report

Chapter 4: Resources Used

In this chapter, the resources that have been used during the project are explained. These resources range from Python libraries to pre-trained CNN models and even hardware or cloud resources.

4.1 Base Code

The code for the Cloud API is based on the face detection code in [31]. This code has been used as a template and has been completely modified to fit the requirements of this project, adapting several functions and creating many others to provide the API services and Kubernetes integration.

The code for the Edge nodes is based on the code in [32]. This code has been used to handle the detection of faces with a fast model. More code has then been added in order to send the extracted faces to the Cloud API. Also, some demo programs have been prepared based on this code, for instance: to add persons to the database from a webcam or a file; to recognize in real-time from a webcam, a file or a video stream; and to assess the efficiency and performance of the system. These programs use the cloud functions to handle the face recognition.

4.2 Libraries – Cloud API

Many libraries have been used for the development of this project. The ones that do not come with the base Python 3 installation can be found in the following subsections, where both the functions and the use that is given to them is explained.

4.2.1 NumPy

The NumPy library [33] is a packet for mathematical operations and scientific computing for Python. It is open source and can be used freely without restrains under the BSD license.

In this project, the NumPy library is used for many mathematical calculations, data type conversions and data storage variable types. It is especially useful as it provides a range of variables and functions that are not available within the base Python 3 mathematical functions.

4.2.2 Pillow

The Pillow library [34] is a packet for image handling, processing and storage based on the well-known Python Imaging Library (PIL). It can be used for tasks like reading or storing images, and some basic image processing.

In this project, the Pillow library is used to read and write image files, as well as to handle image rotations.

4.2.3 Facenet PyTorch

The Facenet PyTorch library [35] provides pretrained Inception ResNet V1 models in PyTorch, as well as an implementation of MTCNN also in PyTorch. Both these features are of great relevance for this project, as they provide key functionality required to run both the face extraction (from the source images) and the face embedding generation.

4.2.4 Torch and TorchVision

The PyTorch library [36] provides a way to deal with tensors and dynamic neural networks in a simple fashion. It provides tensor computation capabilities and deep neural networks functions.

The TorchVision packet [37] provides access to commonly used datasets, model architectures and image transforms that can be leveraged in computer vision applications.

In this project, the PyTorch library is used in conjunction with the TorchVision library to transform the tensors used in the project to different sizes and to obtain lists of the different images in the database used to train the SVM classifier.

4.2.5 OpenCV Python

The OpenCV library [38] provides a series of tools for developers and researchers of computer vision and machine learning software. It includes thousands of algorithms that can be integrated in the different applications that use it.

In this project, the OpenCV library is used to perform many operations regarding image processing for the pre-processing pipeline where the image is optimized prior to its input to the embedding generation CNN.

4.2.6 Scikit Learn

The Scikit Learn library [39] provides tools for predictive data analysis. These predictions can use one of the many prediction mathematical models and methods available within the library.

In this project, the Support Vector Machine (SVM) predictor is used in the shape of a Linear SVM kernel for multiclass classification. This is used in the classification part, where the embedding generated by the feature extractor is fed into the Linear SVM classifier to obtain the Face ID to which the face in the image being processed pertains.

4.2.7 Joblib

The Joblib library [40] provides a set of tools to pipeline and to cache functions, storing them on disk. This allows for faster execution of recurrent functions and avoiding re-evaluation of commands.

In this project, Joblib is used to store the face recognition pipeline, that is later used by the API by simply loading the job instead of creating the same resources again and again. This allows to speed up the processing of single images in one-image batches.

4.2.8 Flask and Flask RESTPlus

The Flask library [41] is used in conjunction with the Flask RESTPlus library [42] to allow easy development and implementation of APIs with additional option to use Swagger [43] documentation.

In this project, both the Flask and Flask RESTPlus libraries are used to build the API calls and functions that allow the system to handle all the types of requests required to service the different maintenance and usage HTTP requests.

4.2.9 uWSGI

The uWSGI packet [44] allows Python APIs to run in a server with multithreading and all the necessary tools. This allows the HTTP request to be handled by uWSGI and, therefore, to be stored in queues and dispatched to the different threads of execution.

The uWSGI server has been used in this project to dispatch the HTTP requests that are received by each Pod to the different processing threads available in that Pod, while queuing the requests that cannot be handled immediately.

4.2.10 Werkzeug

The Werkzeug library [45] is a WSGI web application library. It provides a series of tools that can be used in the Python code that is being run to interact with the WSGI server.

In this project, the Werkzeug library is used to deal with the images that are received by the uWSGI server via HTTP Post requests. It provides a series of data structures that can be used to receive files in the Flask API.

4.2.11 Requests

The Requests library [46] provides a way to perform HTTP requests without complex socketing. It can perform all types of HTTP requests and can send many types of data annexed to them.

In this project, the Requests library is used to send the results of the face recognition and extraction to the external Mobile APP API when required.

4.2.12 MySQL Connector

The MySQL Connector library for Python [47] provides a way to connect to SQL databases and send all types of queries to these databases. The queries can both obtain and store information.

In this project, the MySQL Connector library is used to connect to a database which stores information regarding the people that the system can recognize and the detections of these people.

4.2.13 Matplotlib

The Matplotlib library [48] aids with the visualization of data in Python. It does so by providing functions to display data on the screen in meaningful ways such as graphs and plots.

In this project, the Matplotlib library has been used during the development stages for the visualization of images and data, which helps to visualize the changes to the images in the processing and facial extraction pipelines.

4.3 Libraries – Edge Processing

In the Edge Processing program, some libraries are used, including some that overlap with the ones used in the Cloud API, such as OpenCV and NumPy. The ones that do not come incorporated into the base Python 3 installation can be found in the following subsections.

4.3.1 OpenCV

The OpenCV library [38] has already been described in the section corresponding to the Cloud API.

In this project, OpenCV is used in the Edge Processing to capture the video received in the node, extract the frames, resize the images, change the color palette and some other things.

4.3.2 NumPy

The NumPy library [33] has already been described in the section corresponding to the Cloud API.

In this project, NumPy is used in the Edge Processing for many mathematical operations relating to the images, the handling of detection scores and other uses.

4.3.3 ONNX

The Open Neural Network Exchange (ONNX) library [49] is a standard for machine learning interoperability that allows the storage and representation of machine learning models.

In this project, ONNX is used in the Edge Processing to read the CNN model (Ultra-Light) used to detect the location of the faces in the input image extracted from the video streams received by the Edge Node.

4.3.4 Requests

The Requests library [46] has already been described in the section corresponding to the Cloud API.

In this project, the Requests library is used in the Edge Processing to send the face images extracted from the input video feeds to the Cloud API for further processing.

4.4 Hardware and Cloud Resources

For prototyping purposes, an HP DL360e Server with 32 threads and 96 Gb of RAM running VMWare ESXi has been used to run some containers, Kubernetes, NFS and DB tests.

For development of the application and for the final implementation and testing, an Azure deployment has been used, with many servers running the Kubernetes cluster, the NFS storage server and the DB server.

4.5 Builds Pipeline

Regarding container builds, the pipeline that has provided automated builds for this project is composed of a GitHub private repository with all the source code connected to a Docker Hub account, in which a private repository compiles and releases builds as new commits are sent to the GitHub.

Chapter 5: IMPLEMENTATION

To achieve the design objectives previously explained, the facial recognition architecture has been implemented both locally for the Edge Computing functions and in the Azure Cloud for the Cloud API.

In this chapter, both the program implementations of the architecture described as well as the program flows and API request routines for some common maintenance and model update tasks are included.

5.1 CAMERAS

In a real deployment, the video streams processed by the edge computing nodes would come from a series of cameras deployed on-site. However, as this is not a production deployment but rather a test deployment, the cameras have been substituted by pre-recorded video streams (live video stream could also be used).

The video streams are stored in local storage in the computer used as an edge computing node. These video feeds are then read by the edge computing program in the same way as they would be read if the video were to be coming from an online stream (from an IP camera).

5.2 EDGE COMPUTING

The edge computing nodes carry out the extraction of the faces included in the different frames of the video feeds coming from the cameras. This extraction is performed with a lower accuracy when compared with the re-extraction in the cloud, as processing speed is key and false positives are corrected in the Cloud API re-extraction.

Each edge computing processing node only requires the execution of one thread of the frame and face extraction program per each video feed that requires to be processed simultaneously (i.e., one thread per camera).

As in this test deployment the cameras have been substituted by video files, the number of threads is not limited by the number of cameras available.

To achieve the task at hand, the program that runs in the edge node acts as shown in Figure 11, connecting to the video stream in the setup phase, and extracting the frames until the program is terminated or the video stream stops. The extracted frames are then processed by a face detection algorithm that provides the position and size of the detected faces in the image. This information is then used to extract the faces from the original image. Once the faces have been extracted, they are then sent to the cloud API for further processing and identification.

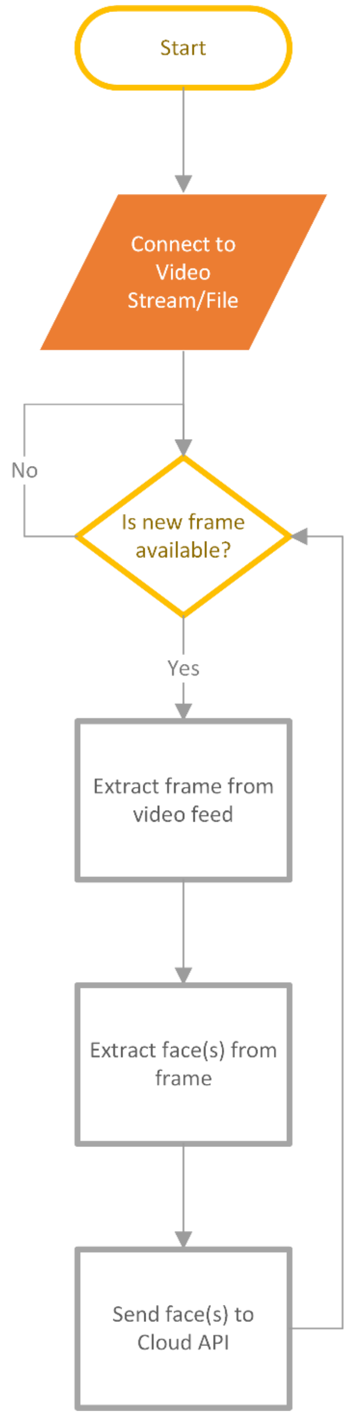

Start
Connect to Video Stream/File
No
Is new frame available?
Yes
Extract frame from video feed
Extract face(s) from frame
Send face(s) to Cloud API

Figure 11: Flowchart of the Edge Node Video Processing Program

The connection to the video streams, which in the case of the test implementation in this project are actually video files, is performed by the OpenCV library. This library handles everything from opening the connection to closing it, including the obtention of individual frames.

To extract the faces in the input frame, the program uses a model encoded in an ONNX file. This model is a CNN (Ultra-Light) that has been trained to detect faces in an image and provide their position and size.

Finally, for every face that is detected, its image is extracted and sent to the Cloud API via an HTTP POST request using the Requests library. When the image has been processed by the Cloud, the detection information is stored in the detections table in the Cloud database.

5.3 Azure Configuration

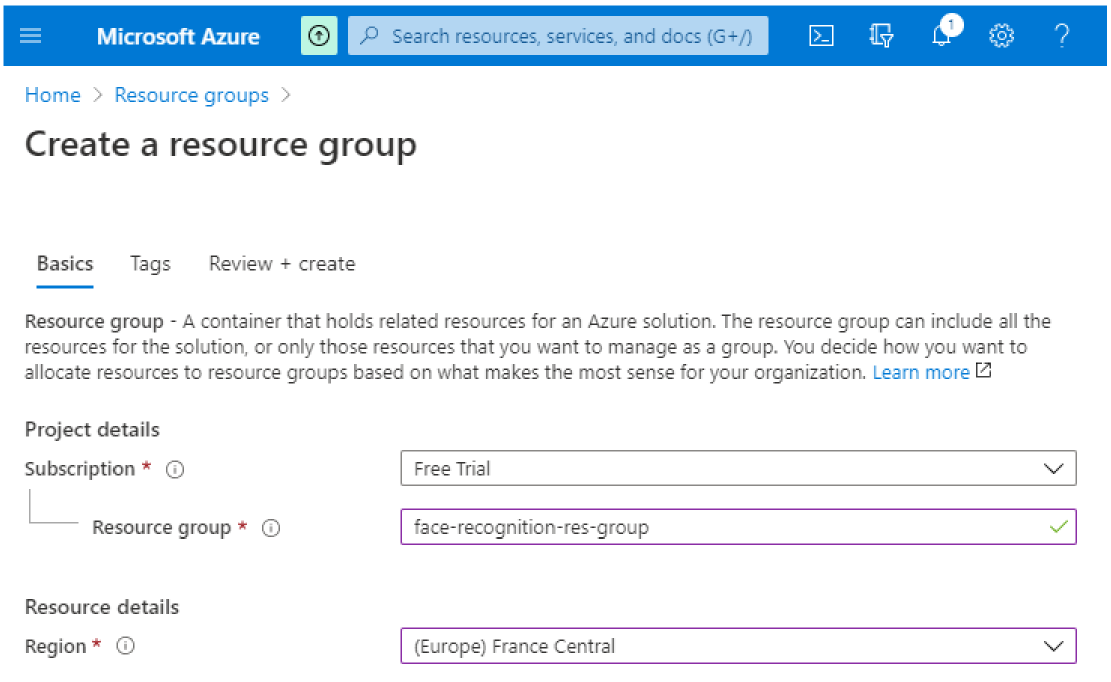
In the following subsections, the creation and setup of the resources required to run the Kubernetes facial recognition application on the Azure cloud are explained.

5.3.1 Create Resource Group

The first step in the process is to create a resource group. In Microsoft's Azure, a resource group is defined as "*a collection of resources that share the same lifecycle, permissions, and policies*".

The use of resource groups allows to join the resources of a project under the same group. This poses multiple benefits from the maintenance, DevOps and billing standpoints.

In this project, a resource group called "face-recognition-res-group" is created in the (Europe) France Central datacenter with the configuration shown in Figure 12. This resource group holds all the resources required for the project (Kubernetes cluster, load balancers, virtual machines, etc.).



Resource group - A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. Learn more

Project details

Subscription *	Free Trial
Resource group *	face-recognition-res-group

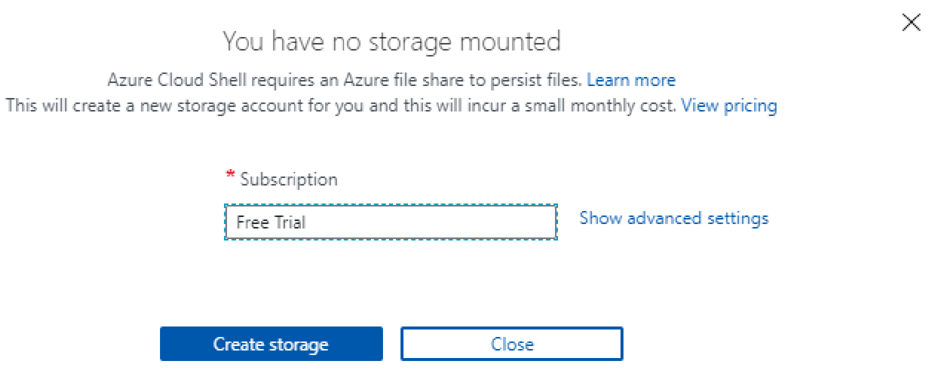
Resource details

Region *	(Europe) France Central

Figure 12: Face Recognition Resource Group Creation

5.3.2 Create Cloud Shell

There are many ways to access the resources and services in the Azure cloud. In this project they are accessed through the Azure Cloud Shell. This allows to run Bash or PowerShell commands to interact from any place with the resources hosted in the cloud. The first time the console is accessed, a storage drive is created within a new default resource group, as shown in Figure 13.


You have no storage mounted

Azure Cloud Shell requires an Azure file share to persist files. Learn more
This will create a new storage account for you and this will incur a small monthly cost. View pricing

* Subscription
Free Trial

Figure 13: Azure Cloud Shell Initial Storage Setup

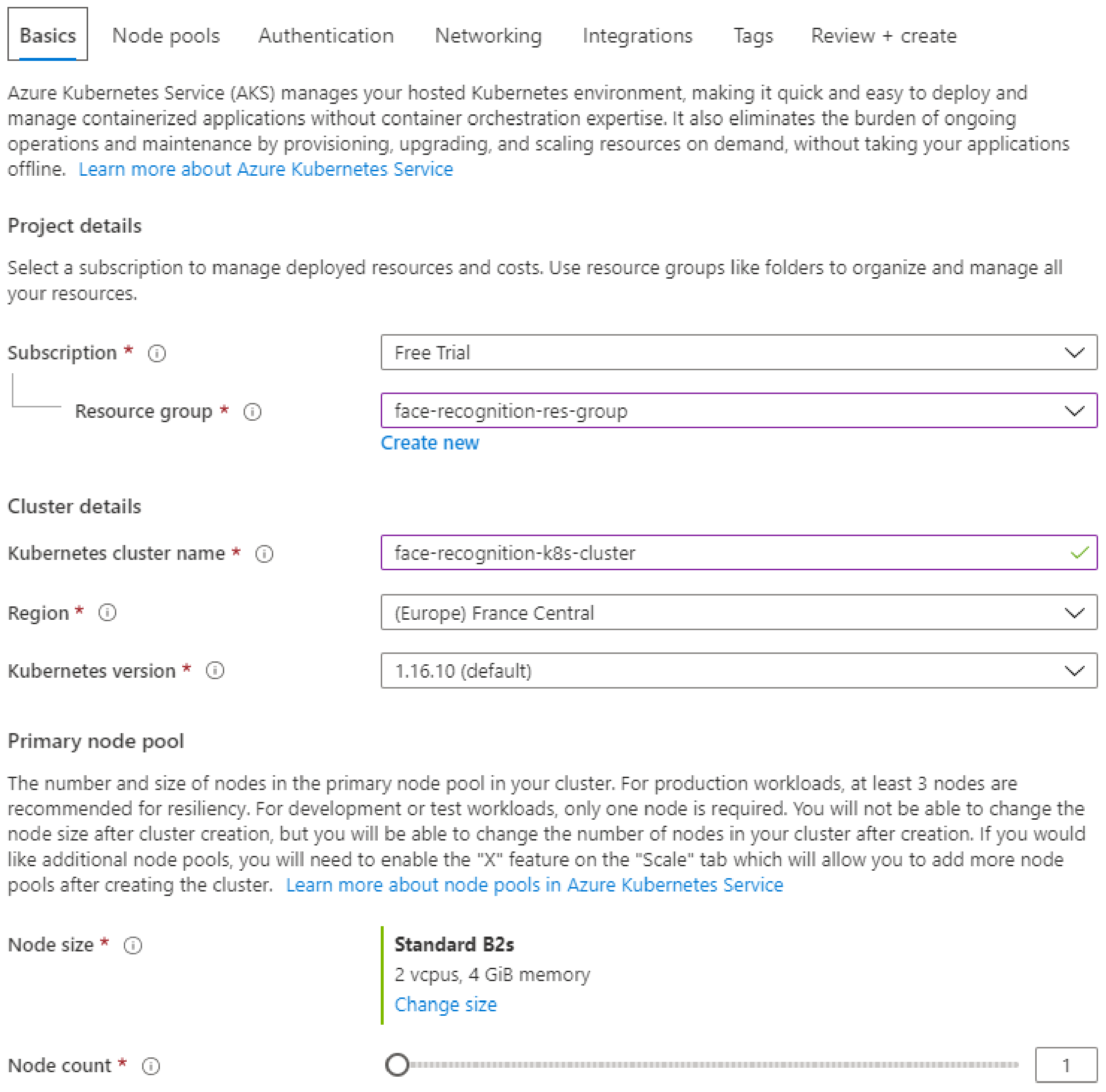
5.3.3 Create Kubernetes Cluster

The second step is to create a Kubernetes cluster which will run the Kubernetes Services, Deployments and Pods required for this system.

This cluster is part of the resource group created in the previous step. It runs in the (Europe) France Central Azure datacenter. The Kubernetes version installed into the nodes in the primary node pool is 1.16.10.

The system nodes are based on the Standard B2ms node type, which is powered by 2 Virtual CPU (VCPU) cores and 8 GiB of RAM per node. For testing purposes, more nodes have been added. In one of the tests, 3 Standard E2s_v3 nodes were used. These nodes provide each 2 VCPUs and 16 GiB of RAM. The minimum settings to run the system are with the system node count set to 1. In a different deployment, the node type and number would be set to different values according to the requirements of the specific system.

The name of the Kubernetes cluster is set to face-recognition-k8s-cluster. All the configuration is introduced into the Azure wizard as shown in Figure 14. There are more options that could be modified in other setup pages, although they are omitted in this document.



Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline. Learn more about Azure Kubernetes Service

Project details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *: Free Trial

Resource group *: face-recognition-res-group
Create new

Cluster details

Kubernetes cluster name *: face-recognition-k8s-cluster

Region *: (Europe) France Central

Kubernetes version *: 1.16.10 (default)

Primary node pool

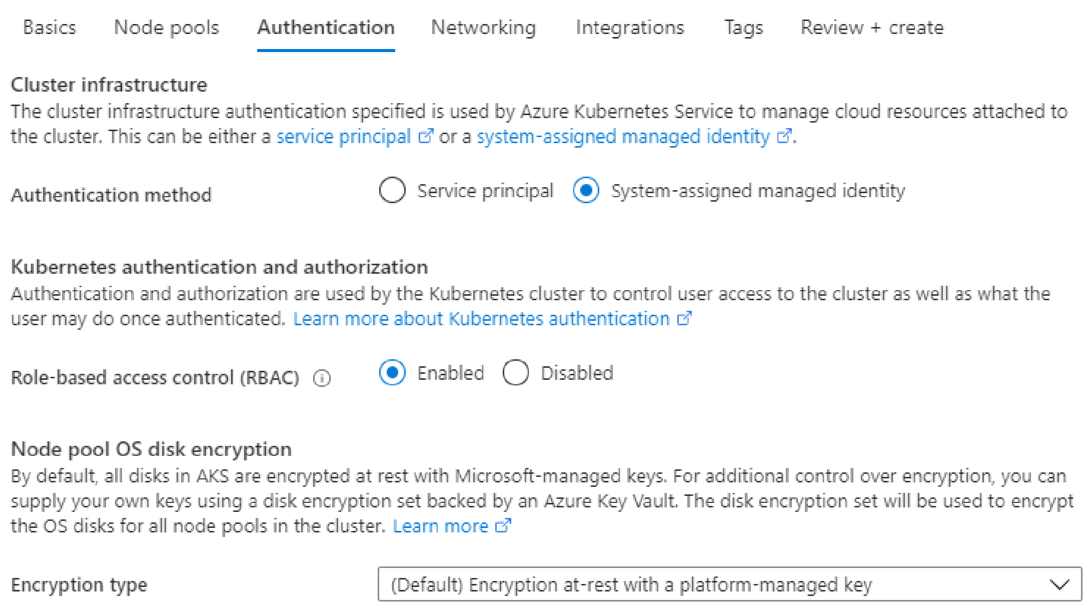
The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. You will not be able to change the node size after cluster creation, but you will be able to change the number of nodes in your cluster after creation. If you would like additional node pools, you will need to enable the "X" feature on the "Scale" tab which will allow you to add more node pools after creating the cluster. Learn more about node pools in Azure Kubernetes Service

Node size *: **Standard B2s**
2 vcpus, 4 GiB memory
Change size

Node count *: 1

Figure 14: Azure Kubernetes Cluster Creation

In the Authentication tab, the Authentication method is changed from a “service principal” to a “system-assigned” managed identity, leaving the parameters as shown in Figure 15. This instructs the Azure cloud to generate a managed identity (login credentials) for the resource being created, which will be follow the lifecycle of the resource (i.e., will be removed if the resource is ever removed and will be automatically renewed when required). Afterwards, the cluster can be created.



Basics | Node pools | **Authentication** | Networking | Integrations | Tags | Review + create

Cluster infrastructure
The cluster infrastructure authentication specified is used by Azure Kubernetes Service to manage cloud resources attached to the cluster. This can be either a service principal or a system-assigned managed identity.

Authentication method	Service principal [✓ System-assigned managed identity]

Kubernetes authentication and authorization
Authentication and authorization are used by the Kubernetes cluster to control user access to the cluster as well as what the user may do once authenticated. Learn more about Kubernetes authentication

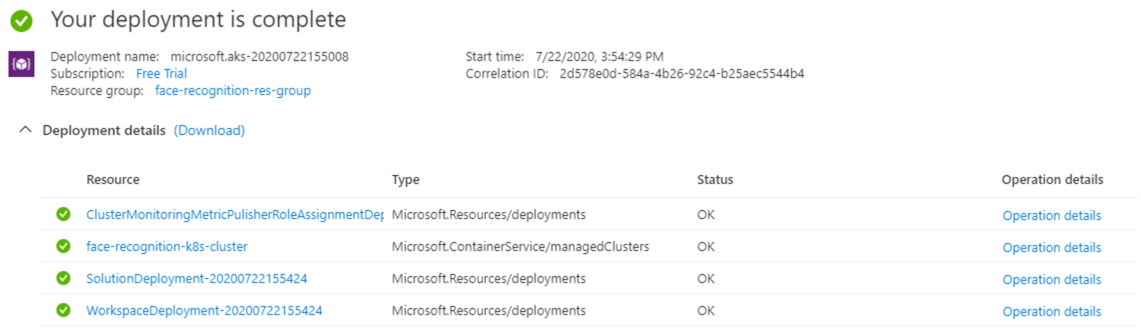
Role-based access control (RBAC)	[✓ Enabled] Disabled

Node pool OS disk encryption
By default, all disks in AKS are encrypted at rest with Microsoft-managed keys. For additional control over encryption, you can supply your own keys using a disk encryption set backed by an Azure Key Vault. The disk encryption set will be used to encrypt the OS disks for all node pools in the cluster. Learn more

Encryption type	(Default) Encryption at-rest with a platform-managed key

Figure 15: Azure Kubernetes Cluster Authentication Configuration

Once the deployment of the cluster and monitoring metrics for the Azure Insights shown in Figure 16 has finished, the cluster can be added to the cloud shell for management.



Your deployment is complete

Deployment name: microsoft.aks-20200722155008
Subscription: Free Trial
Resource group: face-recognition-res-group
Start time: 7/22/2020, 3:54:29 PM
Correlation ID: 2d578e0d-584a-4b26-92c4-b25aec5544b4

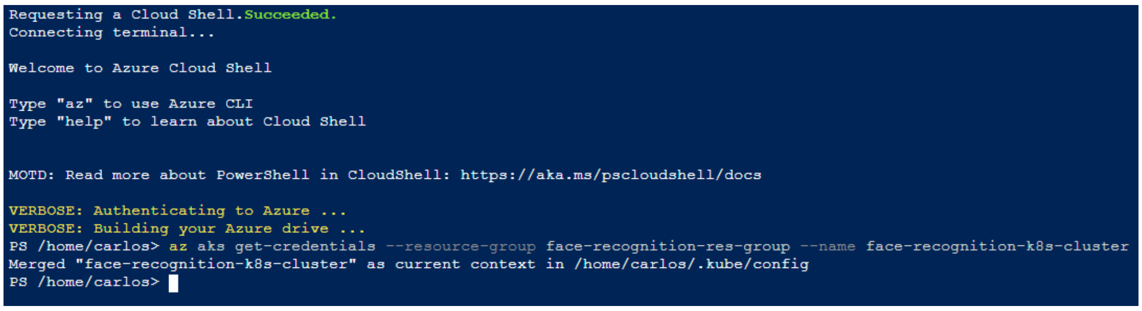
Deployment details (Download)

	Resource	Type	Status	Operation details
✓	ClusterMonitoringMetricPulisherRoleAssignmentDe	Microsoft.Resources/deployments	OK	Operation details
✓	face-recognition-k8s-cluster	Microsoft.ContainerService/managedClusters	OK	Operation details
✓	SolutionDeployment-20200722155424	Microsoft.Resources/deployments	OK	Operation details
✓	WorkspaceDeployment-20200722155424	Microsoft.Resources/deployments	OK	Operation details

Figure 16: Result of the Kubernetes Cluster Creation

5.3.4 Add Kubernetes Cluster to Cloud Shell

To manage the cluster from the Azure Cloud Shell, the credentials have been pulled with the get-credentials command, as shown in Figure 17. This command gets the credentials for a given resource and stores them in the cloud shell to allow the access to this resource from the shell.

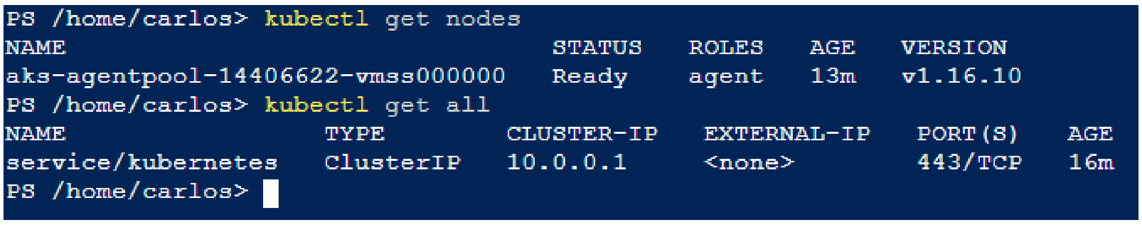


```
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

Welcome to Azure Cloud Shell

Type "az" to use Azure CLI
Type "help" to learn about Cloud Shell


MOTD: Read more about PowerShell in CloudShell: https://aka.ms/pscloudshell/docs

VERBOSE: Authenticating to Azure ...
VERBOSE: Building your Azure drive ...
PS /home/carlos> az aks get-credentials --resource-group face-recognition-res-group --name face-recognition-k8s-cluster
Merged "face-recognition-k8s-cluster" as current context in /home/carlos/.kube/config
PS /home/carlos> 
```

Figure 17: Addition of the Kubernetes Cluster to the Azure Cloud Shell

5.3.5 Check Setup

As a final step, the status of the nodes, Services, Deployments and Pods can be checked to make sure that everything is running as it should. The interaction with the Kubernetes cluster is performed via the “kubectl” command tool. To obtain the health information, two kubectl commands have been executed (“kubectl get nodes” and “kubectl get all”) in the Azure Cloud Shell, as shown in Figure 18.



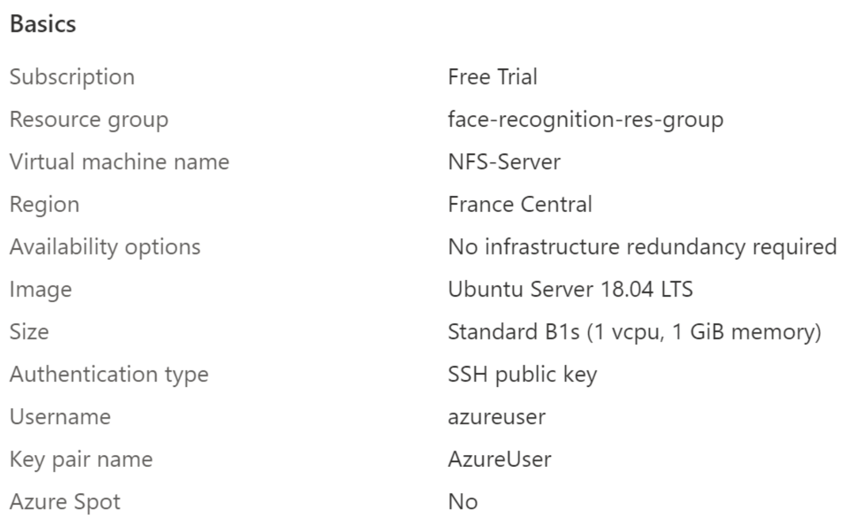
```
PS /home/carlos> kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-agentpool-14406622-vmss000000   Ready    agent   13m   v1.16.10
PS /home/carlos> kubectl get all
NAME                   TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes     ClusterIP   10.0.0.1     <none>        443/TCP   16m
PS /home/carlos>
```

Figure 18: Check of the Setup of the Kubernetes Cluster in the Azure Cloud Shell

5.3.6 NFS Storage VM

The NFS server is hosted in a Virtual Machine (VM). This VM is independent of the Kubernetes deployment and therefore keeps the data safe from any problems with Pods. Note that this can also be implemented in Kubernetes with something called persistent volumes, although it can be more of a hassle to keep in a prototyping environment.

The configuration of this machine, as shown in Figure 19, is in a 1 VCPU, 1 Gb of RAM VM. The machine is assigned to the Azure Kubernetes Service (AKS) virtual network to allow the group to access the VM without routing. The machine is configured to run the Ubuntu Server OS ver. 18.04 Long Term Service (LTS).



Basics

Subscription	Free Trial
Resource group	face-recognition-res-group
Virtual machine name	NFS-Server
Region	France Central
Availability options	No infrastructure redundancy required
Image	Ubuntu Server 18.04 LTS
Size	Standard B1s (1 vcpu, 1 GiB memory)
Authentication type	SSH public key
Username	azureuser
Key pair name	AzureUser
Azure Spot	No

Figure 19: NFS Server Basic Configuration Information

To be able to access the VM from its public IP, the SSH port for the machine must be opened in the security group to which the VM has been added (the AKS security group). Besides, the internal IP address of the VM is made static so that it can be configured in the Kubernetes Deployment YAML file.

5.3.7 Database Server

In this project, the database server is run in a Kubernetes Pod, so no extra configuration is required.

5.4 API Design

The Cloud API that runs on Azure has been designed with modularity, scalability, and adaptability in mind, with a special focus on reliability and high availability. Because of this, the architecture has been designed using Kubernetes deployments and Kubernetes Services.

The general overview of the cloud system, as found in Figure 20, is composed of a Kubernetes cluster, a storage server for the dataset and the machine learning models storage (in the Dataset and Models folders), and a database for storage of the people and their detections in the different cameras and locations (in the People and Detections DB tables).

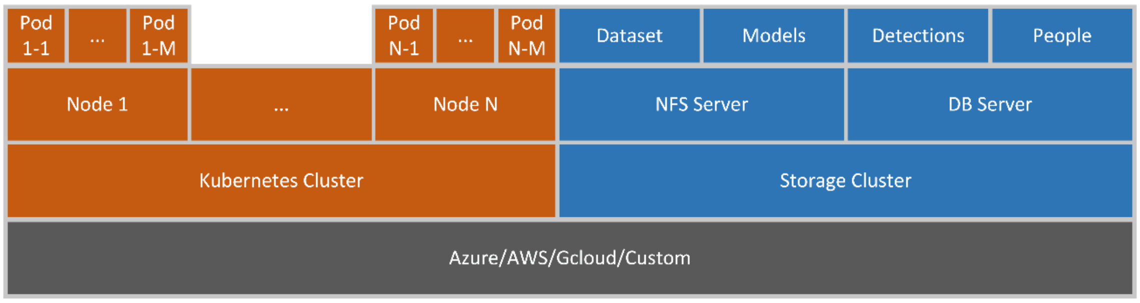


Figure 20: Overview of the Cloud Servers Structure

The information stored in the NFS server is accessed by the replicas of the Kubernetes deployment to train the embedding SVM classifier in the case of the Dataset folder, and to store and retrieve the latest SVM classifier and embedding generator from the Models folder, with the file structure shown in Figure 21.

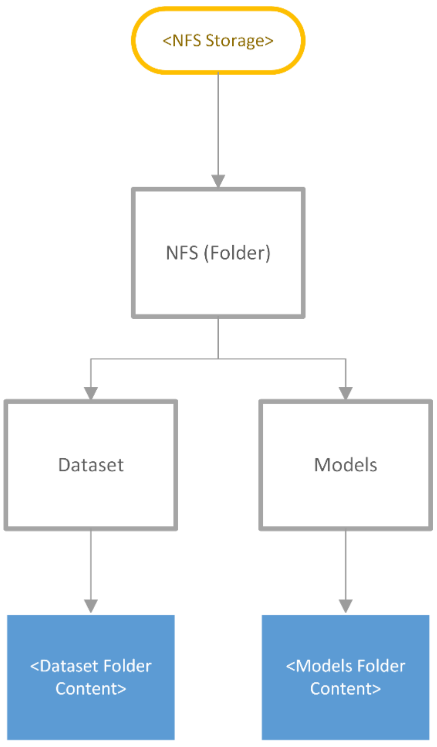


Figure 21: NFS Storage File Structure

The structure of the Dataset folder, as shown in Figure 22, has all the images used to train the classifier in folders with the ID of the person as their name. These folders are created and removed and these images are stored and deleted with the use of Cloud API calls. When the basic training API function is called, these images are used to train the SVM classifier to distinguish between the different people stored in the Dataset folder.

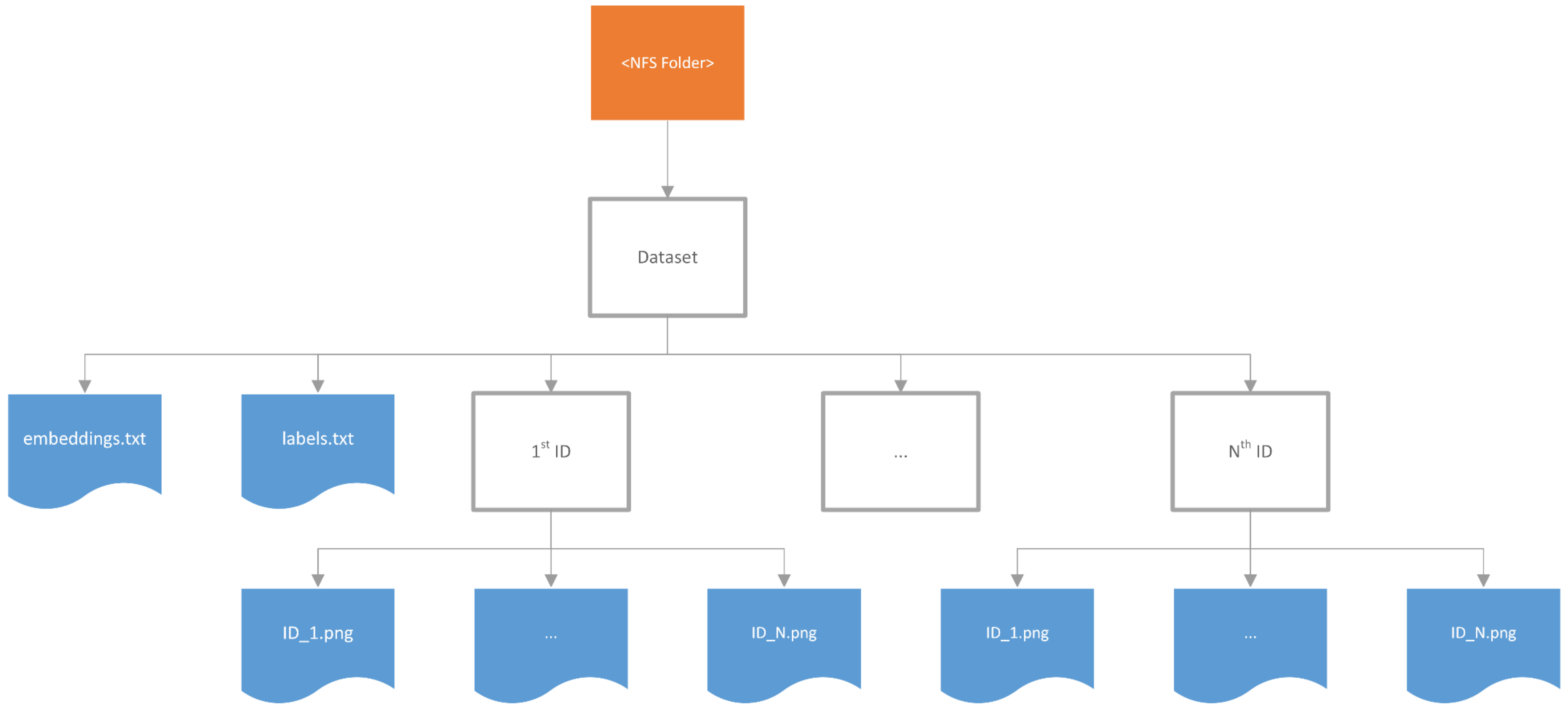


Figure 22: Dataset Folder File Structure

As also seen in Figure 22, there are two other elements stored in the Dataset folder: the embeddings.txt (referred to as Embeddings) and labels.txt (referred to as Labels) files. These files contain the embeddings corresponding to the images in the Dataset folder and the labels that are associated to those embeddings (first embedding in Embeddings file corresponds to the first label in Labels file, etc.).

These files are generated either when the embedding generation API call is used or when the images and IDs are added to the dataset. In fact, whenever one image is added to the dataset, the embedding of this image is generated and appended to the Embeddings file, with the corresponding label appended to the Labels file. When an ID is removed, not only its folder is removed, but also the embeddings and labels associated with that ID.

Keeping the Embeddings and Labels files up to date allows the use of an optimized SVM training method where, instead of going through the Dataset folder generating embeddings for each of the images every time the system is trained, the system can read the Embeddings and Labels files and use them to train the classifier. This is much faster than re-generating the embeddings every time the classifier is trained, especially if only a few images have been added since the last training.

The Models folder contains the models that are generated every time the classifier is trained, as may be seen in Figure 23. The Joblib files containing the SVM classifier and the CNN feature extractor are stored with the time when the training of the SVM classifier started as their name. This data is relevant as the moment the training starts is the moment when the Embeddings and Labels files are read, and therefore the instant in which these files contained the information that was used to train that version of the model.

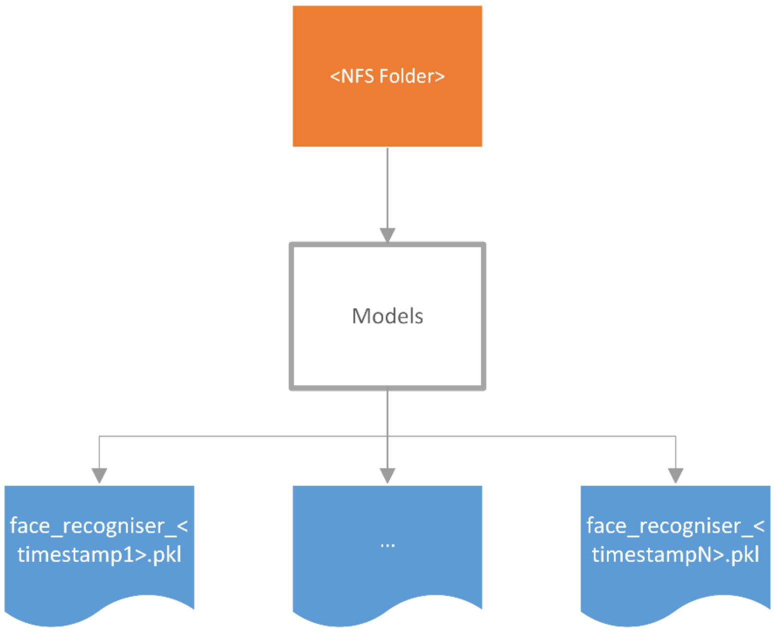

<NFS Folder>
Models
face_recogniser_<timestamp1>.pkl
...
face_recogniser_<timestampN>.pkl

Figure 23: Models Folder File Structure

The database hosted in the database server has two tables as shown in Figure 24: one for the people that can be recognized by the system and one for the detections of the system. The first includes only the name of the person (the Face ID) and an ID as Primary auto-incremental Key, although it is prepared so that more information can be included depending on the specific application of the system.

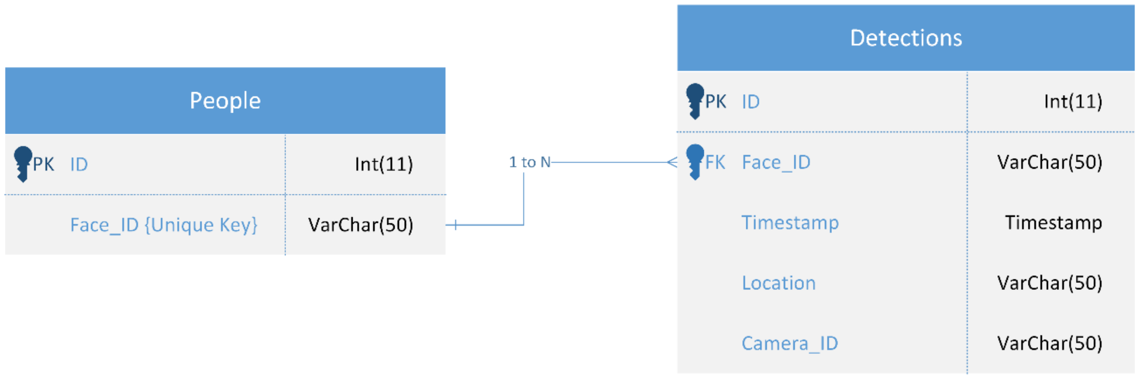


Figure 24: Relational Database Table Structure Diagram

The Detections table contains the time of detection, the location, the camera ID, the Face ID and an ID as Primary auto-incremental Key. This allows to store the most relevant information regarding individual detections, although more information could be added if required by the application.

The Face_ID is a Unique Key in the People table and a Foreign Key in the Detections table. The relationship between the two is of one to many, with one in the People table (as it is a Unique Key) to many in the Detections table.

As seen in Figure 25, when a request comes into the system, it is received by a Load Balancer that decides to which of the Kubernetes replicas of the Kubernetes processing Deployment it sends the request.

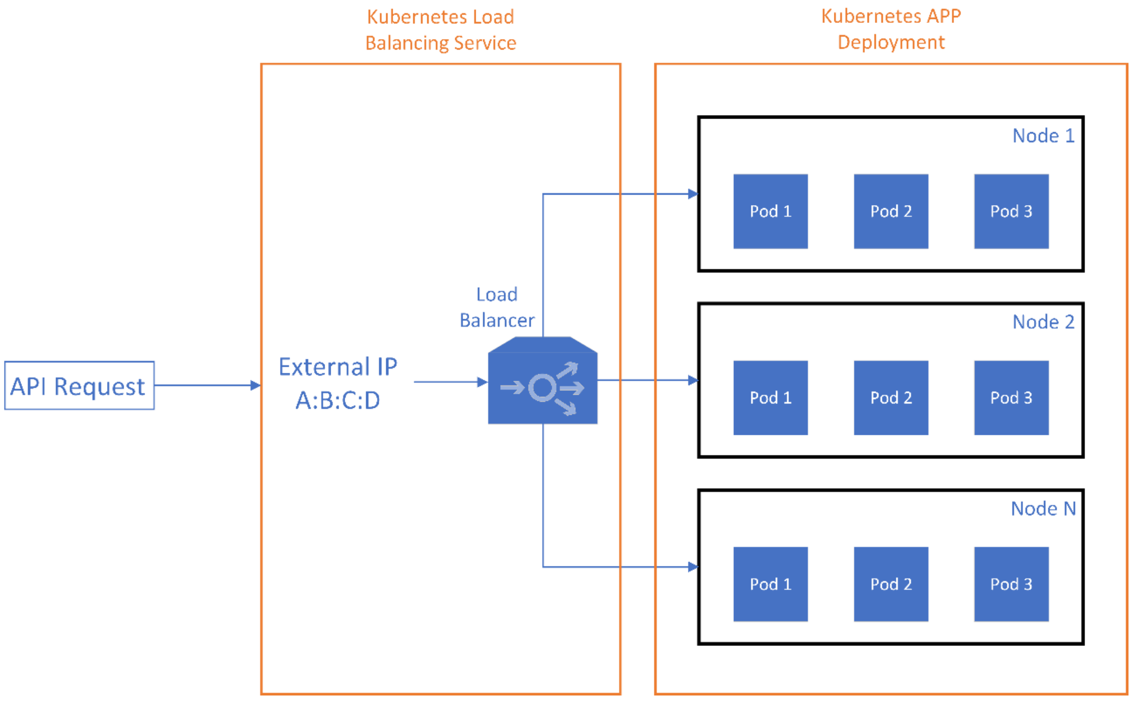


Figure 25: Cloud API Kubernetes Structure and HTTP Request Handling

There are many types of requests that can be handled by the Cloud API. These requests can be classified into five types: dataset modification, model update, face recognition, database querying, and debugging.

In the following subsections, the different types of requests are explained with flowcharts that show the basic actions performed by the Cloud API when one of these requests is performed.

5.4.1 Dataset Query and Update

As previously stated, the Dataset folder contains the images used for training the SVM classifier, stored in folders with the name of the ID to which the images inside pertain.

There are Cloud API calls to see the current state of the Dataset, to Add an ID, to Add an image and to Remove an ID.

An HTTP GET request to the <address>/dataset URL returns the name of all the IDs stored in the Dataset folder along the number of images for each ID. This is accomplished by following the procedure given in Figure 26.

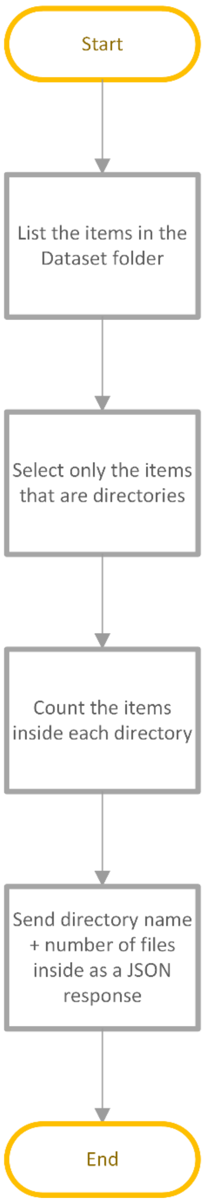

Start
List the items in the Dataset folder
Select only the items that are directories
Count the items inside each directory
Send directory name + number of files inside as a JSON response
End

Figure 26: HTTP GET /dataset API Call Flowchart

An HTTP POST request to the <address>/dataset/<ID> URL adds a new ID to the Dataset folder. When this operation is performed, as shown in Figure 27, a new folder is added to the Dataset and a new person is added to the People table in the database. Some checks are also performed, including security checks to make sure that the folder is created in the expected path.

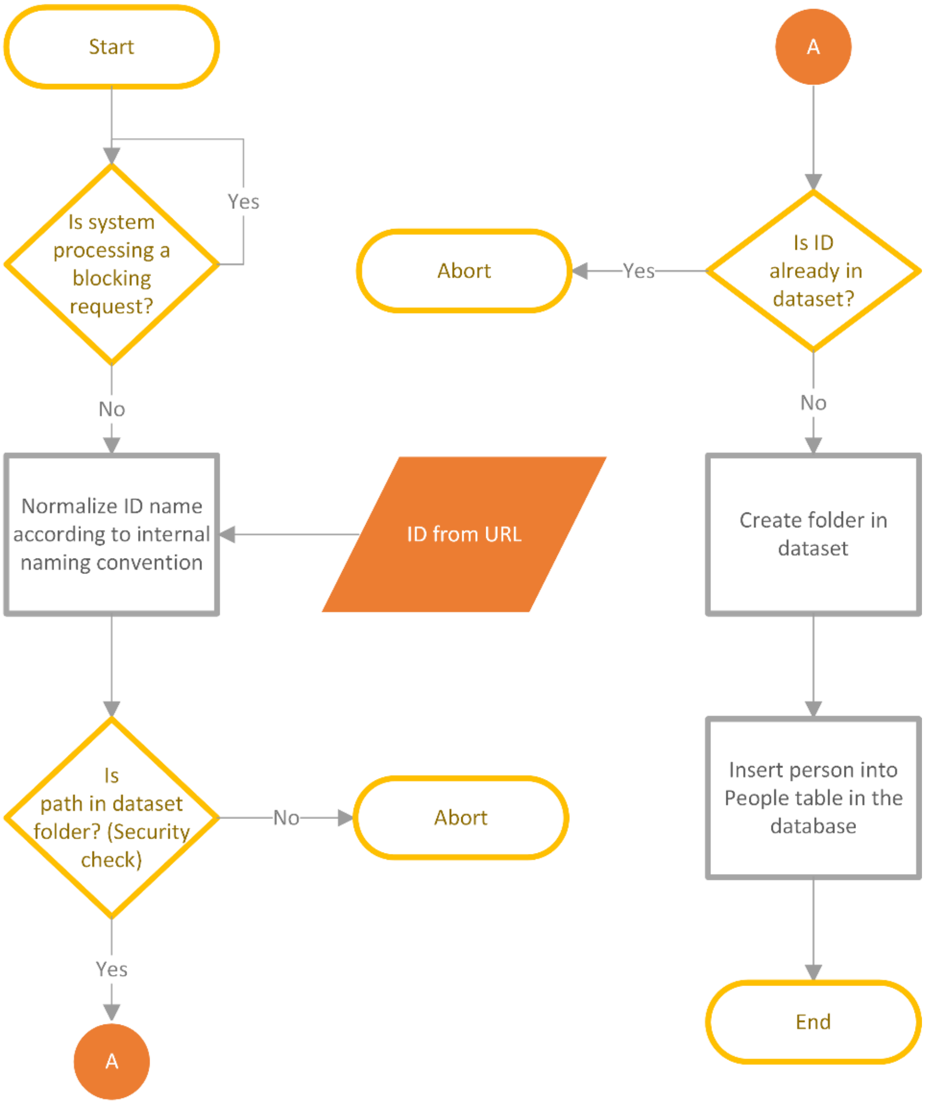
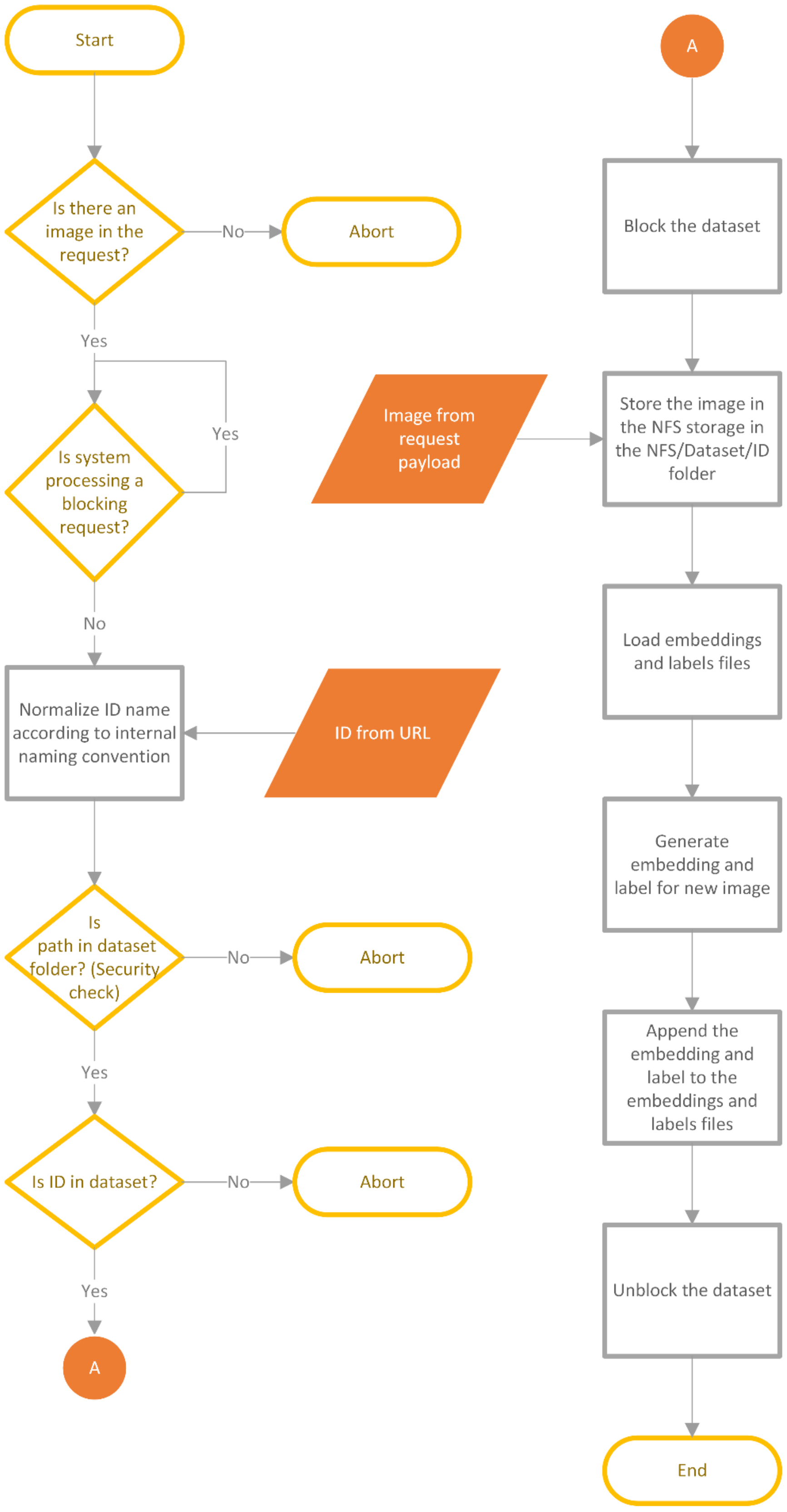

Start
Is system processing a blocking request?
Yes
No
Normalize ID name according to internal naming convention
ID from URL
Is path in dataset folder? (Security check)
No
Abort
Yes
A
A
Is ID already in dataset?
Yes
Abort
No
Create folder in dataset
Insert person into People table in the database
End

Figure 27: HTTP POST /dataset/<ID> API Call Flowchart

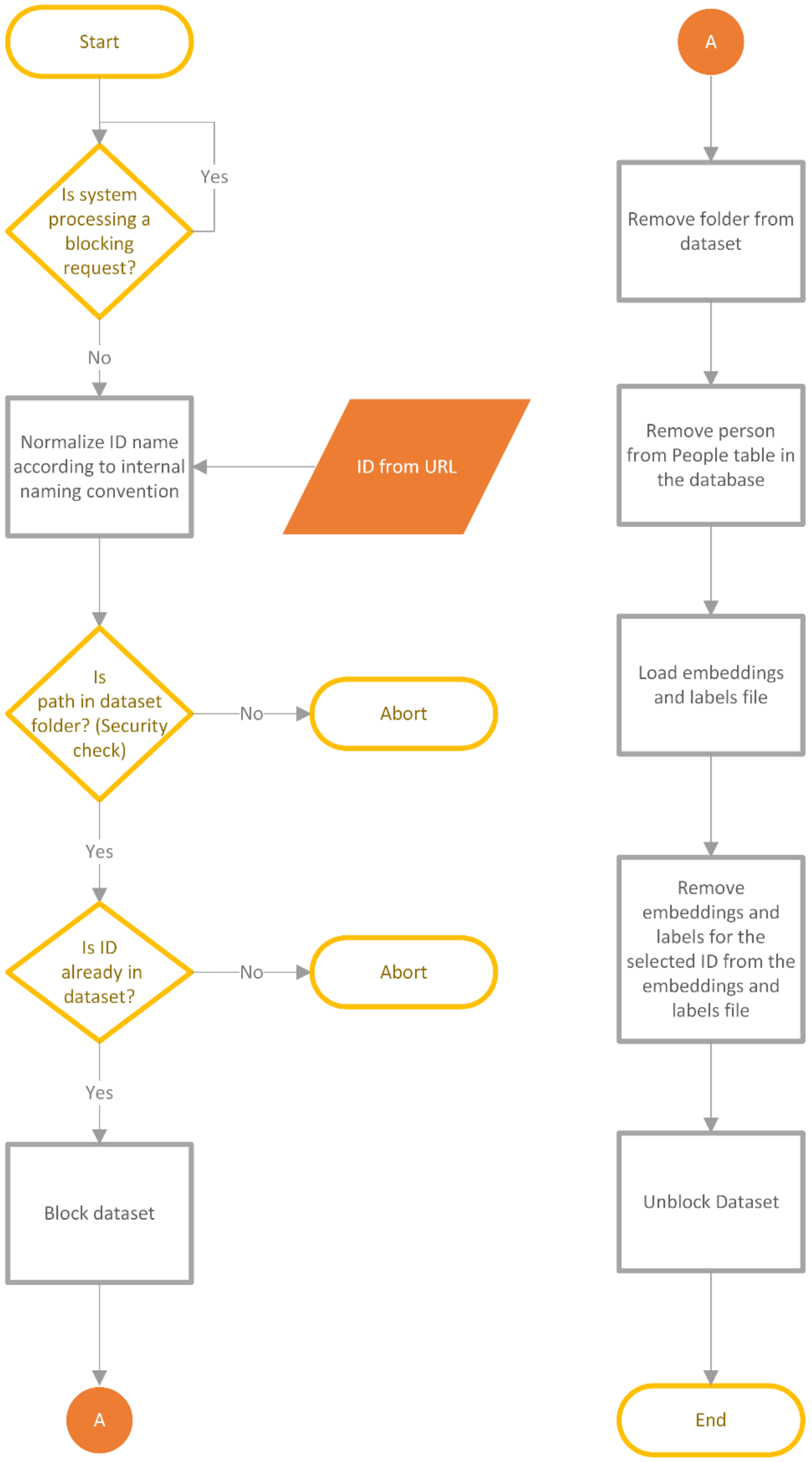
An HTTP POST request to the <address>/dataset/<ID>/image URL that contains an image and a Face ID adds this new image to the ID in the Dataset. This operation not only stores the received image in the Dataset folder, but as shown in Figure 28, also generates the embedding for the image and appends this and the label to the Embeddings and Labels files.



Start
Is there an image in the request?
No
Abort
Yes
Is system processing a blocking request?
Yes
No
Normalize ID name according to internal naming convention
ID from URL
Is path in dataset folder? (Security check)
No
Abort
Yes
Is ID in dataset?
No
Abort
Yes
A
A
Block the dataset
Image from request payload
Store the image in the NFS storage in the NFS/Dataset/ID folder
Load embeddings and labels files
Generate embedding and label for new image
Append the embedding and label to the embeddings and labels files
Unblock the dataset
End

Figure 28: HTTP POST /dataset/<ID>/image API Call Flowchart

An HTTP DELETE request to the <address>/dataset/<ID> URL removes the ID and all the images stored of this ID. Furthermore, as shown in Figure 29, the ID is also removed from the People table in the database and the label and all the embeddings related to this ID are removed from the Embeddings and Labels files.



Start
Is system processing a blocking request?
Yes
No
Normalize ID name according to internal naming convention
ID from URL
Is path in dataset folder? (Security check)
No
Abort
Yes
Is ID already in dataset?
No
Abort
Yes
Block dataset
A
A
Remove folder from dataset
Remove person from People table in the database
Load embeddings and labels file
Remove embeddings and labels for the selected ID from the embeddings and labels file
Unblock Dataset
End

Figure 29: HTTP DELETE /dataset/<ID> API Call Flowchart

5.4.2 Model Update

There are several functions available to update the classification model. If there are images in the Dataset but the Embeddings and Labels files have not been generated, they can be generated by running the "generate embeddings" API call.

If the Embeddings and Labels files are present and a new ID and images are added, or an existing ID is removed or more images are added to an existing ID, the system can be trained with the contents of the Embeddings and Labels files by running the optimized train call. In case the system is to be trained from scratch without considering nor updating the Embeddings and Labels files, the "train" API call can be used.

Starting with the generation of the Embeddings and Labels files, an HTTP POST request to the <address>/model/generate-emb URL creates the Embeddings and Labels files with the images and IDs present in the Dataset, as shown in Figure 30.

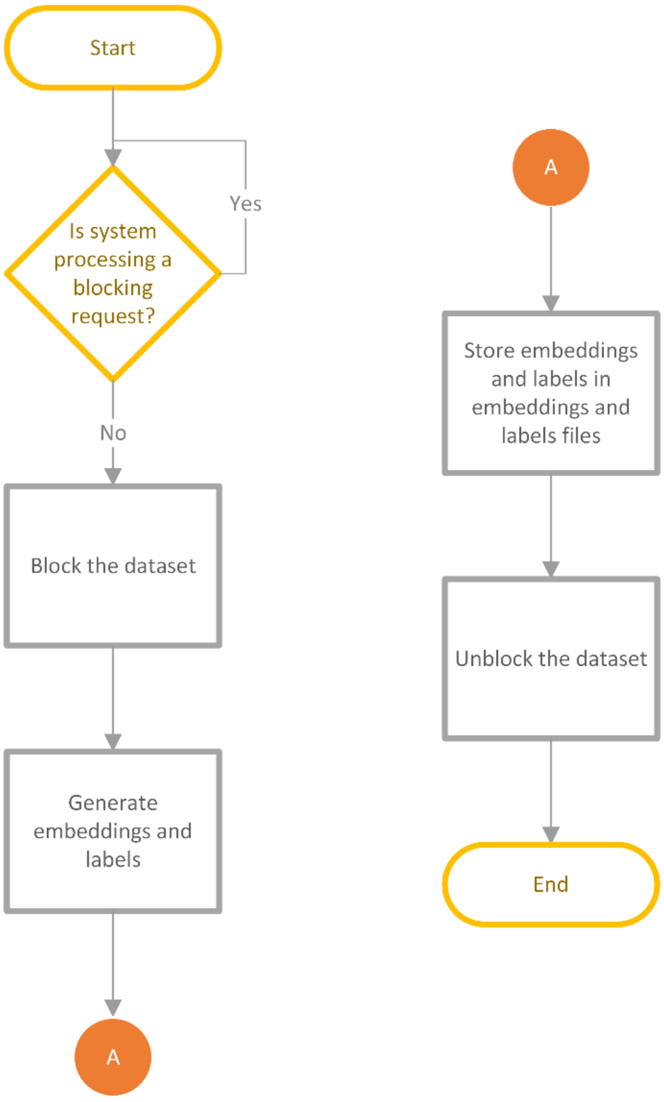

Start
Is system processing a blocking request?
Yes
No
Block the dataset
Generate embeddings and labels
A
A
Store embeddings and labels in embeddings and labels files
Unblock the dataset
End

Figure 30: HTTP POST /model/generate-emb API Call Flowchart

The embedding generation task, as shown in Figure 30 blocks the Dataset as to avoid inconsistencies between the contents of the Embeddings and Labels files and the contents of the dataset folder.

An HTTP POST request to the <address>/model/optimized-train URL, as shown in Figure 31, trains the classifier model with the data available in the Embeddings and Labels files. During this training the Dataset is blocked to avoid modifications to the files while they are being read.

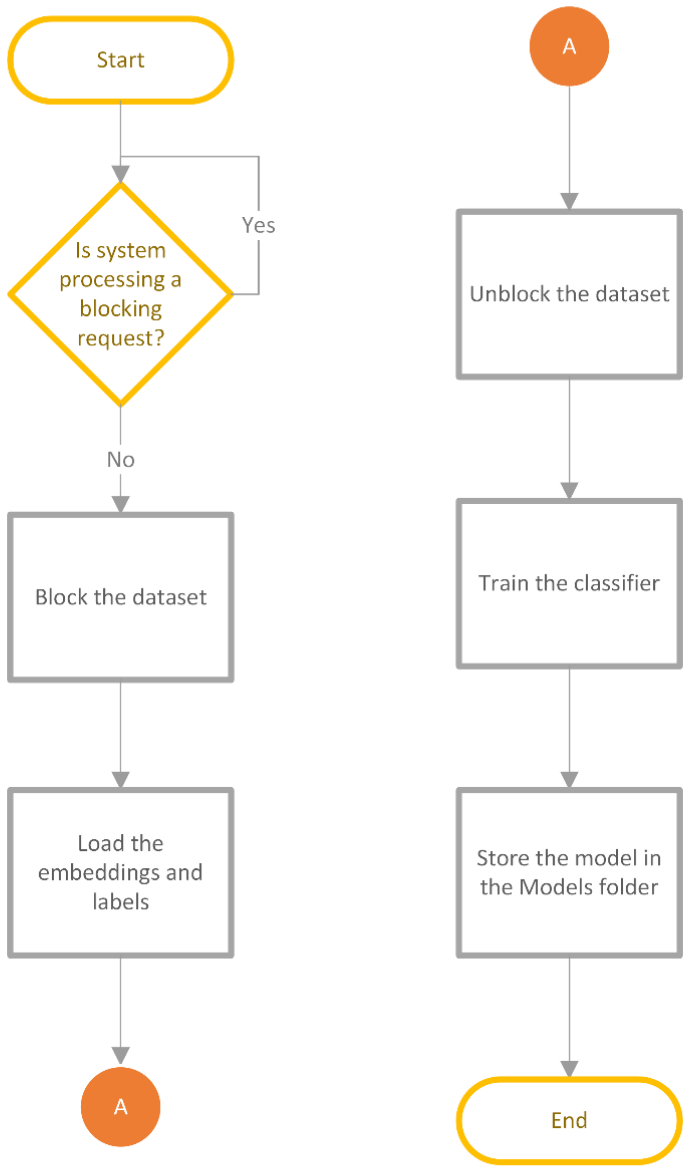

Start
Is system processing a blocking request?
Yes
No
Block the dataset
Load the embeddings and labels
A
A
Unblock the dataset
Train the classifier
Store the model in the Models folder
End

Figure 31:HTTP POST /model/optimized-train API Call Flowchart

An HTTP POST request to the <address>/model/train URL triggers a from-scratch training of the model that, as shown in Figure 32, generates embeddings and labels for all of the images in the dataset but does not store them. These embeddings and labels are then used to train the model. This operation blocks the dataset while the images are being accessed.

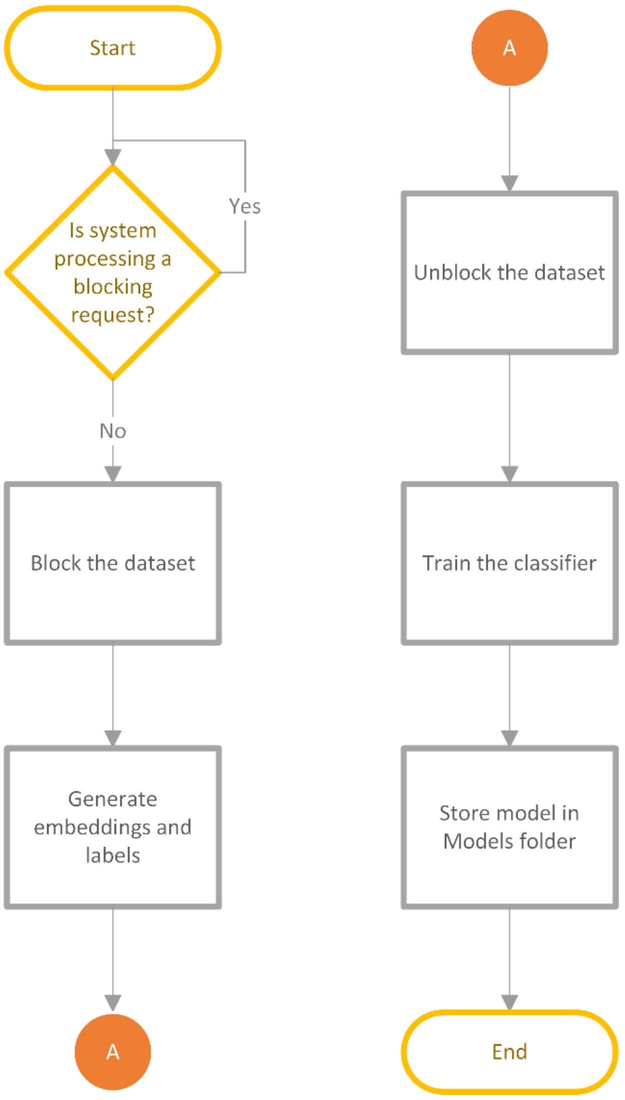
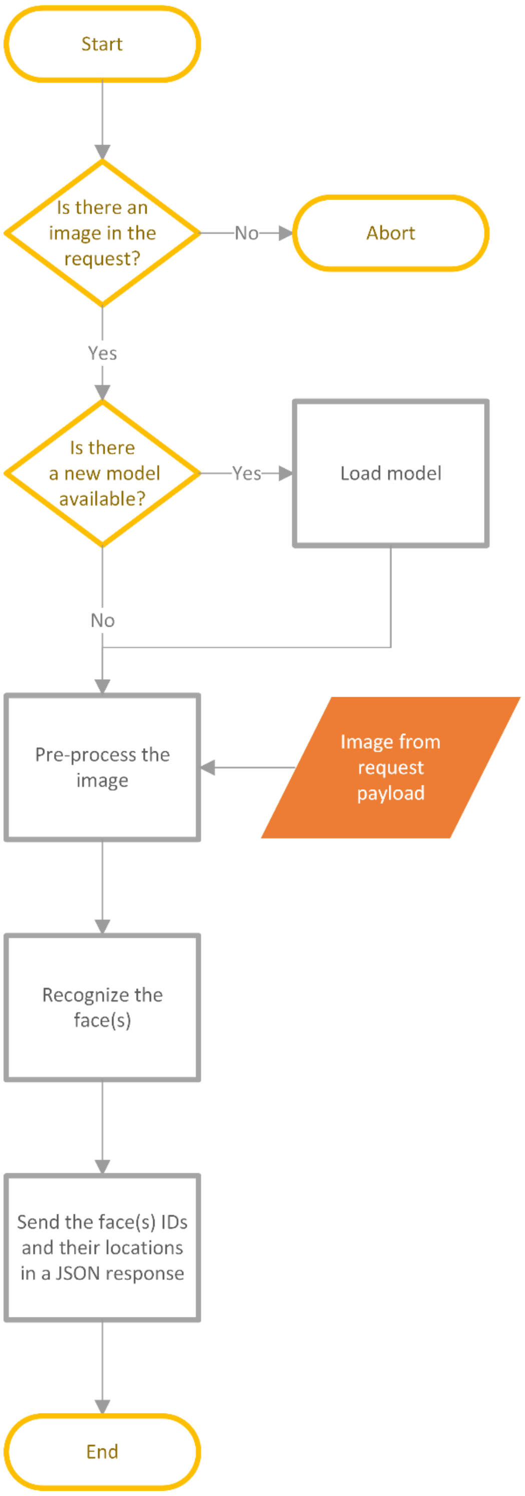

Start
Is system processing a blocking request?
Yes
No
Block the dataset
Generate embeddings and labels
A
A
Unblock the dataset
Train the classifier
Store model in Models folder
End

Figure 32: HTTP POST /model/train API Call Flowchart

5.4.3 Face Recognition

Three functions are provided by the API to perform face recognition with the latest model available. These functions vary in where the detection data is stored or sent to. The processed data can either be sent as a response to the user, sent to an external service or stored in the internal Detections table of the database.

An HTTP GET request to the <address>/face-recognition/get-results URL that contains an image in the request payload returns the face(s') identities in the image, with the probability and location of each one. As shown in Figure 33, the function gets an image as input and returns a JSON with the results.



Start
Is there an image in the request?
No
Abort
Yes
Is there a new model available?
Yes
Load model
No
Pre-process the image
Image from request payload
Recognize the face(s)
Send the face(s) IDs and their locations in a JSON response
End

Figure 33: HTTP GET /face-recognition/get-results API Call Flowchart

An HTTP POST request to the <address>/face-recognition/send-results URL that contains an image, the camera ID, and the location in the request's payload triggers the recognition of the face(s) in the image. As shown in Figure 34, after the recognition is performed, the system sends to a predefined remote server each face detected with the camera ID and the location.

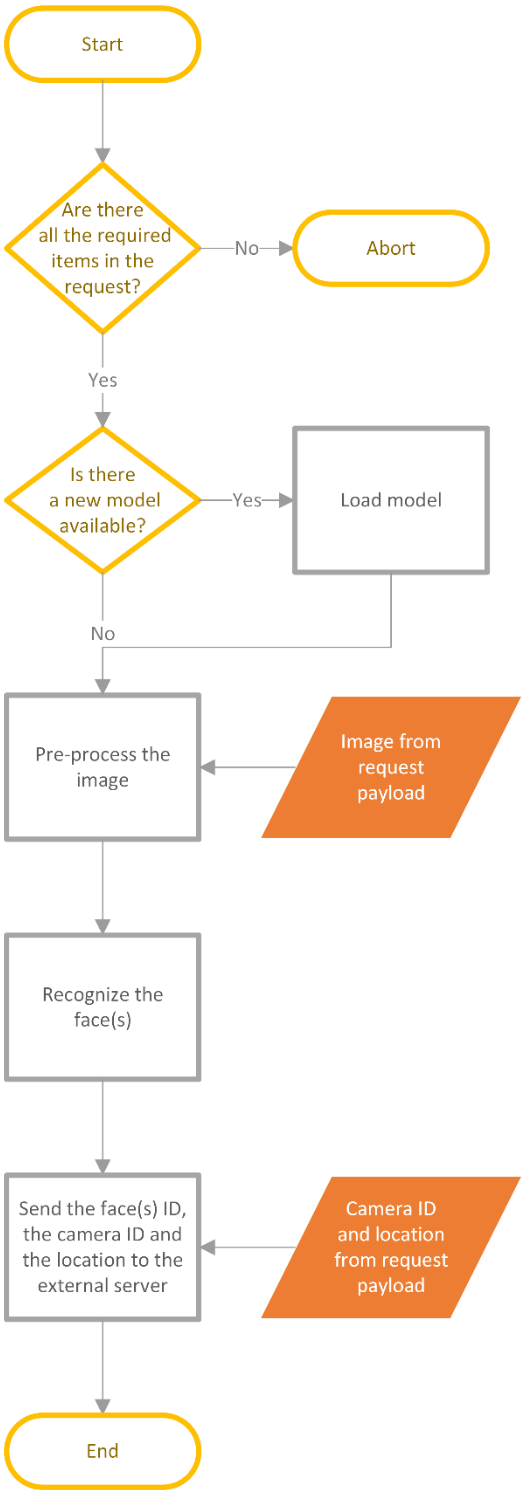

Start
Are there all the required items in the request?
No
Abort
Yes
Is there a new model available?
Yes
Load model
No
Pre-process the image
Image from request payload
Recognize the face(s)
Send the face(s) ID, the camera ID and the location to the external server
Camera ID and location from request payload
End

Figure 34: HTTP POST /face-recognition/send-results API Call Flowchart

Finally, an HTTP POST request to the <address>/face-recognition/store-results URL that contains the image to be processed, the location, and the camera ID triggers the extraction of the face(s) in the image following the procedure in Figure 35. The face(s) recognized in the image is(are) then stored in the detection database with the other data received in the request payload.

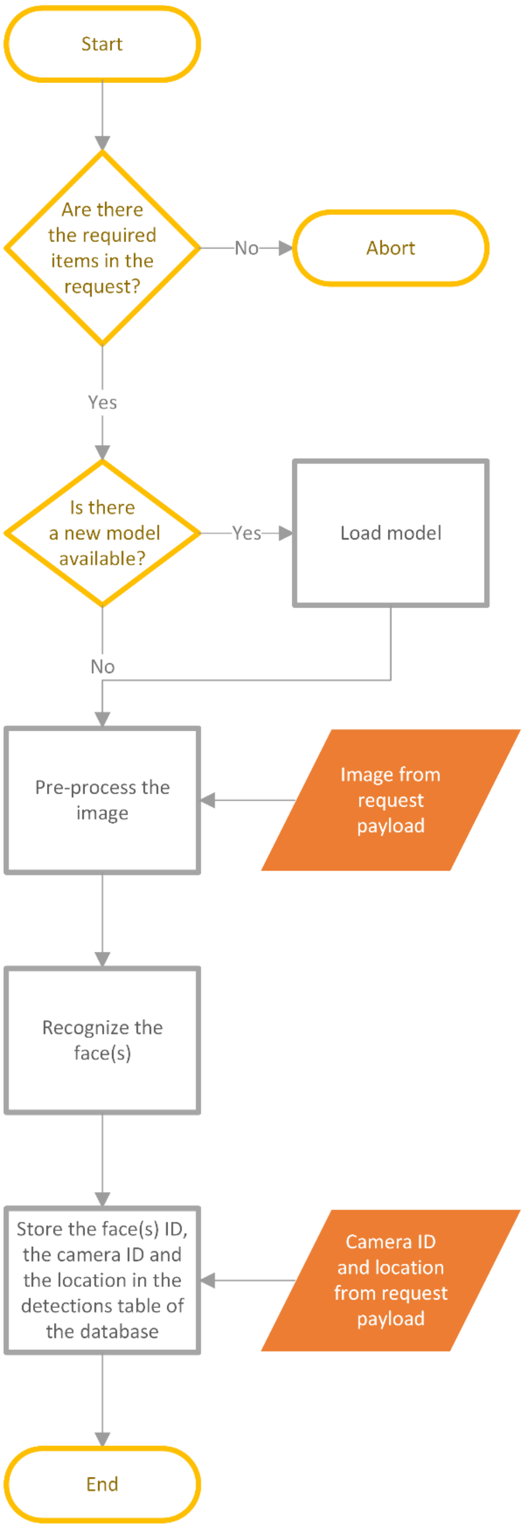

Start
Are there the required items in the request?
No
Abort
Yes
Is there a new model available?
Yes
Load model
No
Pre-process the image
Image from request payload
Recognize the face(s)
Store the face(s) ID, the camera ID and the location in the detections table of the database
Camera ID and location from request payload
End

Figure 35: HTTP POST /face-recognition/store-results API Call Flowchart

The only limit to how many of the get-results, send-results and store-results API calls can be processed simultaneously is imposed by the hardware running the Kubernetes containers. In addition, it is important to remember that there is no downtime when a new model is trained, only a slight delay in the first requests after the model has been trained.

5.4.4 Database Querying

A series of functions have been implemented to retrieve the data stored in the database, although if these data were to be used for any specific application, the requests would probably need to be adapted to the particular needs of the application.

The data stored in the People table of the database can be retrieved with an HTTP GET request to the <address>/database/people URL. The procedure involves an SQL query to the database server as shown in Figure 36, and returns a JSON with the data in the table.

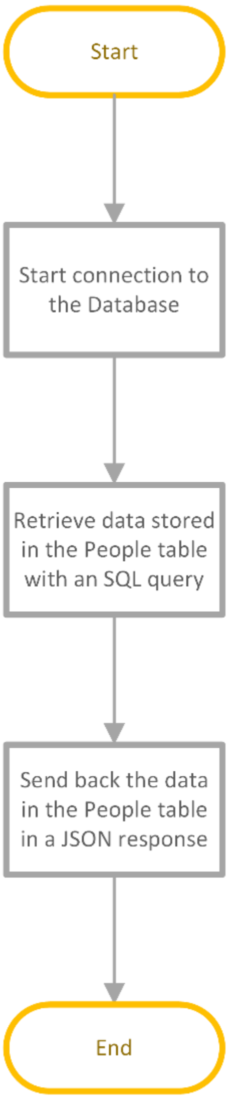

Start
Start connection to the Database
Retrieve data stored in the People table with an SQL query
Send back the data in the People table in a JSON response
End

Figure 36: HTTP GET /database/people API Call Flowchart

An HTTP GET request to the <address>/database/<ID>/last-known-location URL that contains a Face ID returns a JSON with the last location where the person identified by the Face ID was detected, with the camera ID and with the Timestamp of the detection, as shown in Figure 37.

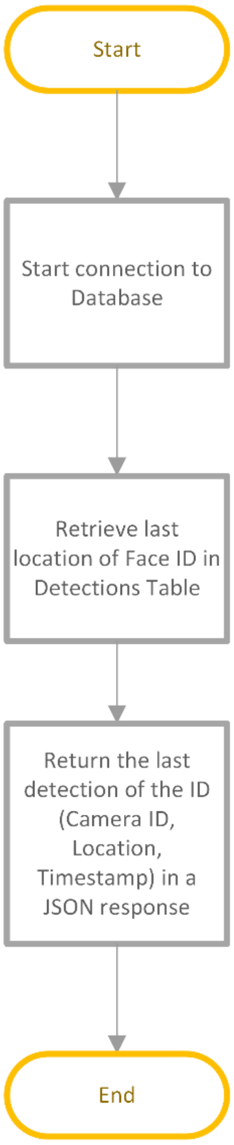

Start
Start connection to Database
Retrieve last location of Face ID in Detections Table
Return the last detection of the ID (Camera ID, Location, Timestamp) in a JSON response
End

Figure 37: HTTP GET /database/<ID>/last-known-location API Call Flowchart

5.4.5 Debugging Requests

Some API calls have been implemented for debugging purposes. These functions are necessary to obtain data regarding the execution of the Cloud API calls and to force some situations in debugging scenarios.

The first of these calls is one to obtain the Cluster IP address of the node executing a given request (in the case of the load balancer Deployment with multiple replicas of the application running concurrently this IP address varies from request to request). It can be obtained with an HTTP GET request to the <address>/debug/IP URL. The details of the procedure performed in the cloud server when this request is received can be found in Figure 38.

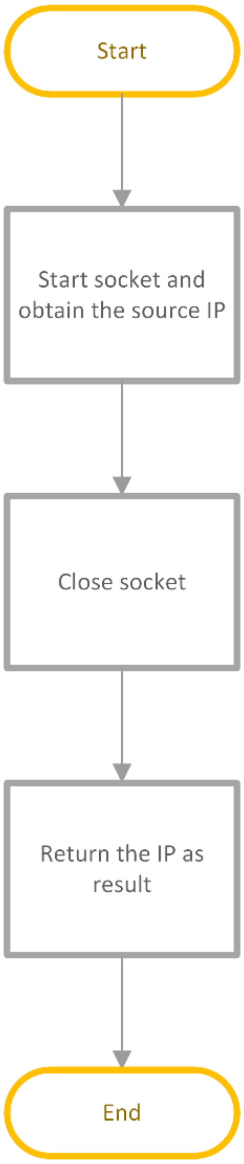


Figure 38: HTTP GET /debug/IP API Call Flowchart

Also, the name of the node executing a request can be obtained by sending an HTTP GET request to the <address>/debug/node-name URL. This triggers the execution of the procedure in Figure 39, which returns the name of the node executing the request.

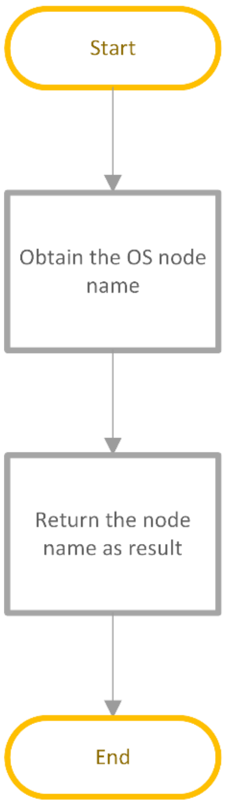


Figure 39: HTTP GET /debug/node-name API Call Flowchart

Sometimes, due to premature node deletion or for testing purposes, the dataset is left blocked. It can be unblocked without accessing the NFS server by sending an HTTP POST request to the <address>/debug/unblock-dataset URL. As shown in Figure 40, this action unblocks the dataset.

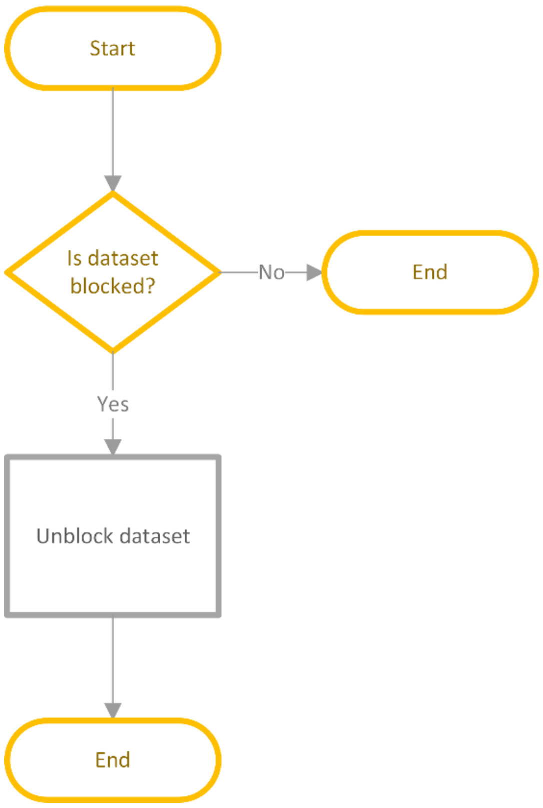


Figure 40: HTTP POST /debug/unblock-dataset API Call Flowchart

5.4.6 Common Procedures

There are some procedures which would be used in a day-to-day basis working with the Cloud API, such as adding someone to the system, removing someone from the system or recognizing someone. The most relevant of these procedures are described in this subsection from the viewpoint of the interaction with the Cloud API.

The procedure to add someone to the system is shown in Figure 41. It consists in adding the Face ID, adding the images one by one to the system and then training the classifier by requesting an optimized training. In this process, the embeddings and labels generated before this procedure are used in conjunction with the ones generated when adding the images to train the multiclass SVM classifier.

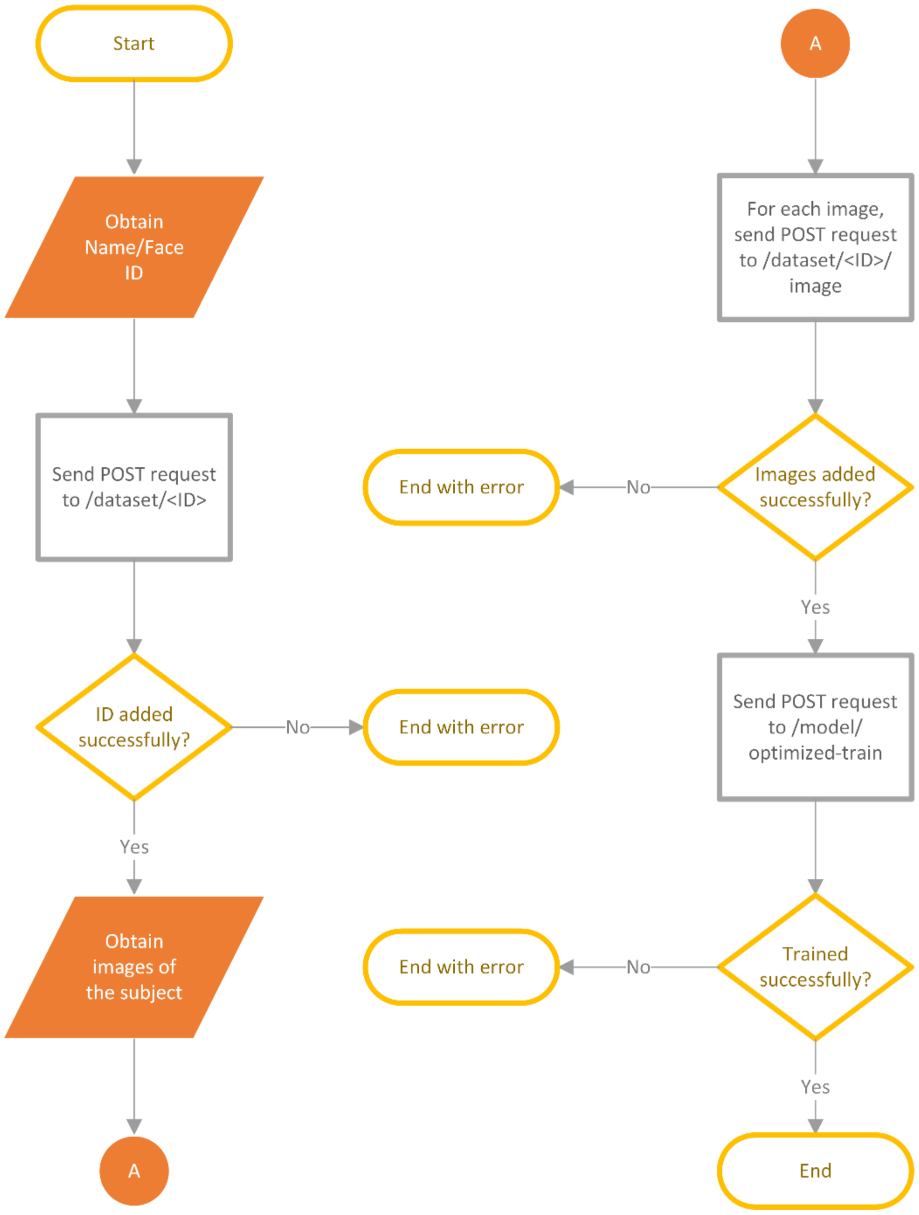

Start
Obtain Name/Face ID
Send POST request to /dataset/<ID>
ID added successfully?
No
End with error
Yes
Obtain images of the subject
A
A
For each image, send POST request to /dataset/<ID>/ image
Images added successfully?
No
End with error
Yes
Send POST request to /model/ optimized-train
Trained successfully?
No
End with error
Yes
End

Figure 41: Add ID, Images and Train Procedure Flowchart

To remove a user from the system, the procedure shown in Figure 42 is used. It consists in requesting the removal of the ID, what will internally remove all the images associated with the ID as well as the images for this ID. This is followed by a request for an optimized training to make the changes effective in the classifier (if adding and/or deleting many IDs and/or images, it is possible and recommendable to perform the optimized only once at the end).

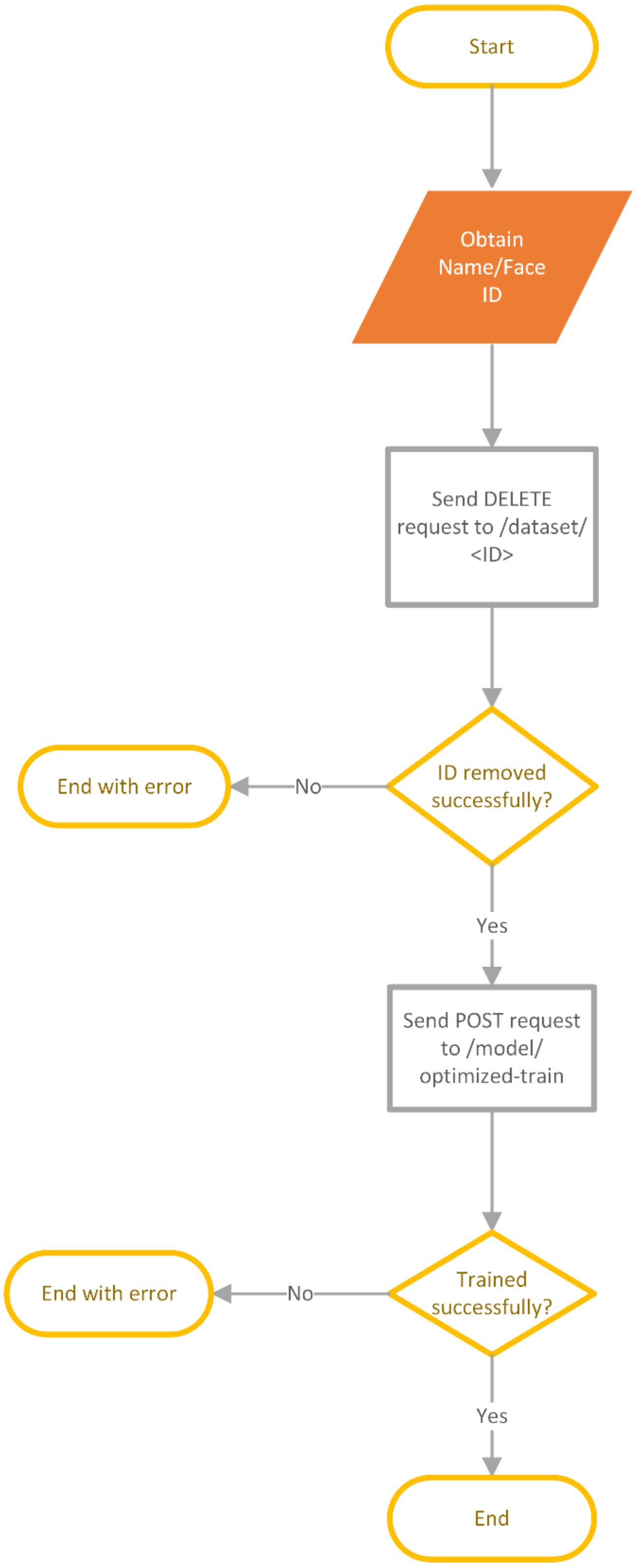
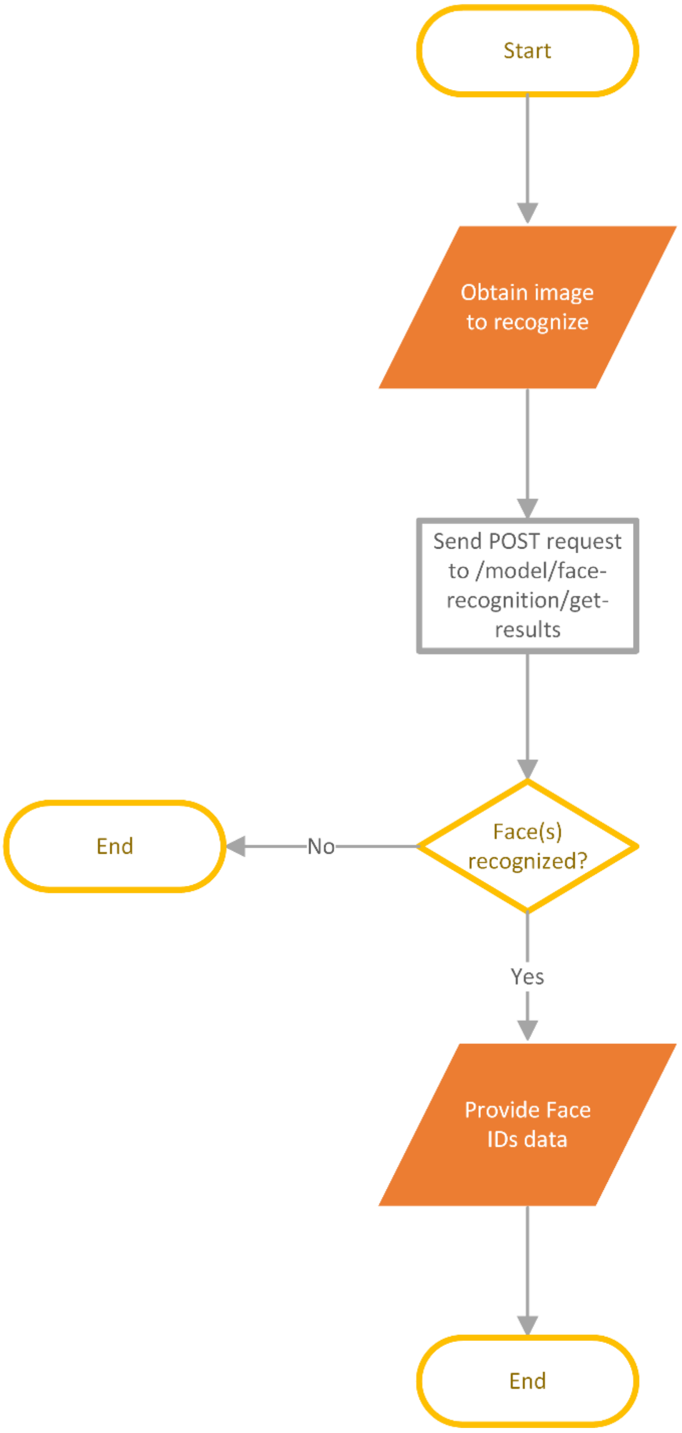


Figure 42: Remove ID (and Images) and Train Procedure Flowchart

To get the identities of the face(s) in an image, the procedure is to send the get-faces request as shown in Figure 43, which returns the identities of the face(s) detected by the Cloud API.



Start
Obtain image to recognize
Send POST request to /model/face-recognition/get-results
Face(s) recognized?
No
End
Yes
Provide Face IDs data
End

Figure 43: Get Face(s) in Image Procedure Flowchart

The procedure in Figure 44 can be used to send the identities of the face(s) in an image to an external server/API. This external API is part of an application that has been implemented by one of the members of the research team within which this project has been developed.

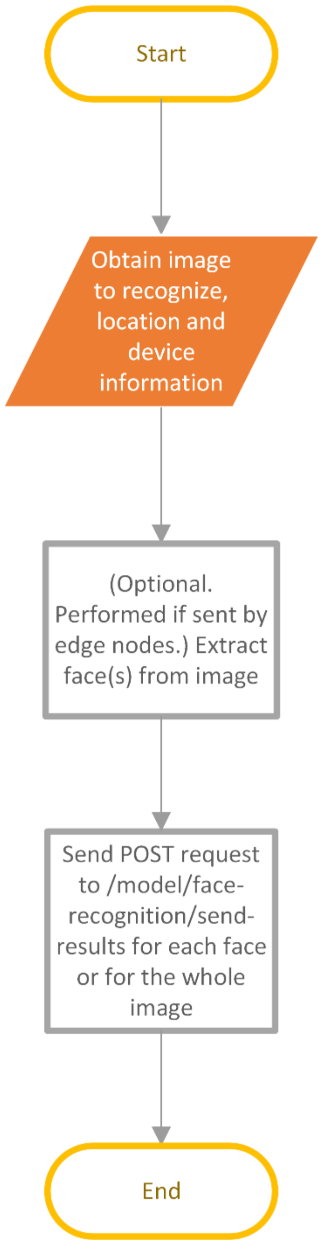

Start
Obtain image to recognize, location and device information
(Optional. Performed if sent by edge nodes.) Extract face(s) from image
Send POST request to /model/face-recognition/send-results for each face or for the whole image
End

Figure 44: Send Face(s) Procedure Flowchart

Finally, the procedure to store the face(s) detected in an image, shown in Figure 45, can be used to store the result of the detections in the Cloud database.

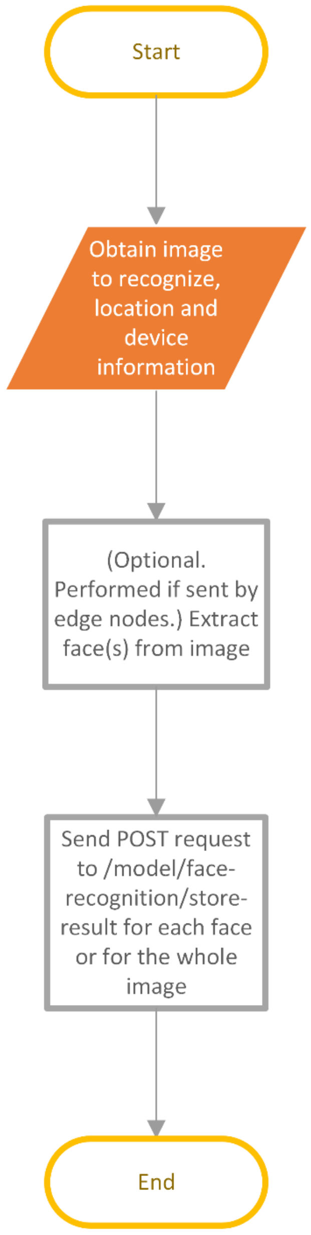

Start
Obtain image to recognize, location and device information
(Optional. Performed if sent by edge nodes.) Extract face(s) from image
Send POST request to /model/face-recognition/store-result for each face or for the whole image
End

Figure 45: Store Face(s) Procedure Flowchart

5.5 API DEPLOYMENT

A requirement to deploy the API into the Cloud Servers, in this case the Microsoft Azure servers, is to have access to a console with control over the Kubernetes nodes of the system. To simplify this task, the Azure cloud Shell has been used.

To deploy the application to the Azure servers, the first task is to get the storage machines working, as they are necessary for the normal operation of the Python code running the Cloud API on the Kubernetes Deployment.

The NFS VM that was previously created has now to be configured to run an NFS server. This requires the following actions: installation of the “nfs-kernel-server” package; creation of the folders to be exported; setting the right access permissions and ownership to the exported folders; configuration of the exports file; and restart of the NFS kernel. After all these tasks have been performed, the Kubernetes application can be deployed.

To deploy the Kubernetes application, the first step is to clone the Github repository with the YAML configuration files for the Deployment to the Azure Cloud Shell storage. Afterwards, the database Pod, the database and face recognition Services and the face recognition Deployment YAML files of the application can be implemented with the kubectl command line tool.

Note: in the case of the test environment in which this has been deployed, the access to the Docker Hub used to build the docker containers has been restricted, setting it up as a private repository. For accessing private repositories, a credential has been added with the “kubectl create secret docker-registry” command.

Chapter 6: INTERACTION PROGRAMS AND RESULTS

A series of programs have been developed to test the Cloud API for its performance and reliability. These programs include raw performance benchmarks as well as real-world use cases.

Programs have also been developed to showcase the capability of the system to add and remove Face IDs at ease without any downtime. This is demonstrated by adding a person while the system is being used to analyze the images of a camera in real-time. This way, a person that appears as unrecognized becomes recognized once the classifier finishes training.

6.1 RAW PERFORMANCE IN REQUESTS PER SECOND

To show the performance of the system, two configurations have been tested: one pod alone and a deployment of 10 replicas. This allows to see how the system scales with more pods and how the number of VCPU cores available constrains the overall system performance.

To test the different systems, concurrent HTTP POST requests are sent simultaneously from a laptop. For this, the GRequests Python library [50] is used. This library allows for asynchronous transmission of HTTP requests and is based on the Resquests [46] and Gevent [51] libraries.

The HTTP POST requests are sent to the address/face-recognition/get-results URL accompanied by a test image from the Labeled Faces in the Wild (LFW) dataset [52], in this example an image with the face of actor Harrison Ford. The image sent in the request is always the same for consistency and comparability of the results.

To measure the performance, the system sends all the requests at once and measures the time to complete all the requests. Afterwards, it checks that the system has processed all the requests correctly and there are no errors. The time that the operation takes to complete, combined with the requests sent, are used to estimate the requests handled per second by the server with a given set of conditions (number of CPU cores, RAM, number of nodes).

In the first case, only one Pod executing the Cloud API code in a 2 v-Core node with 8 Gb of RAM. A program is created to stress test the system by sending a number of HTTP requests simultaneously. As shown in Table 3, by sending many requests at once, the effect of having two threads running in the Pod can be seen, as by sending more than one concurrent request, the system processes double the number of requests per second when compared with only one request.

Number of Requests	Seconds to Complete	Requests per Second
1	0.318	3.144
25	3.428	7.298
50	6.638	7.536
75	9.862	7.606
100	13.162	7.606

Table 3: Relation Between Number of Requests and Requests per Second for One Pod

By running multiple Pods across two Kubernetes nodes enables the system to improve the performance. With a Deployment of 10 replicas in two nodes with a total of 24 Gb of RAM and 4 v-Cores, the system provides the performance given in Table 4. The results do not show double the performance of the example with one Pod mainly for two reasons: the benchmark code is not able to send the such a high number of requests simultaneously very efficiently; the large number of Pods running concurrently has a detrimental effect in the total performance as only 4 cores are available.

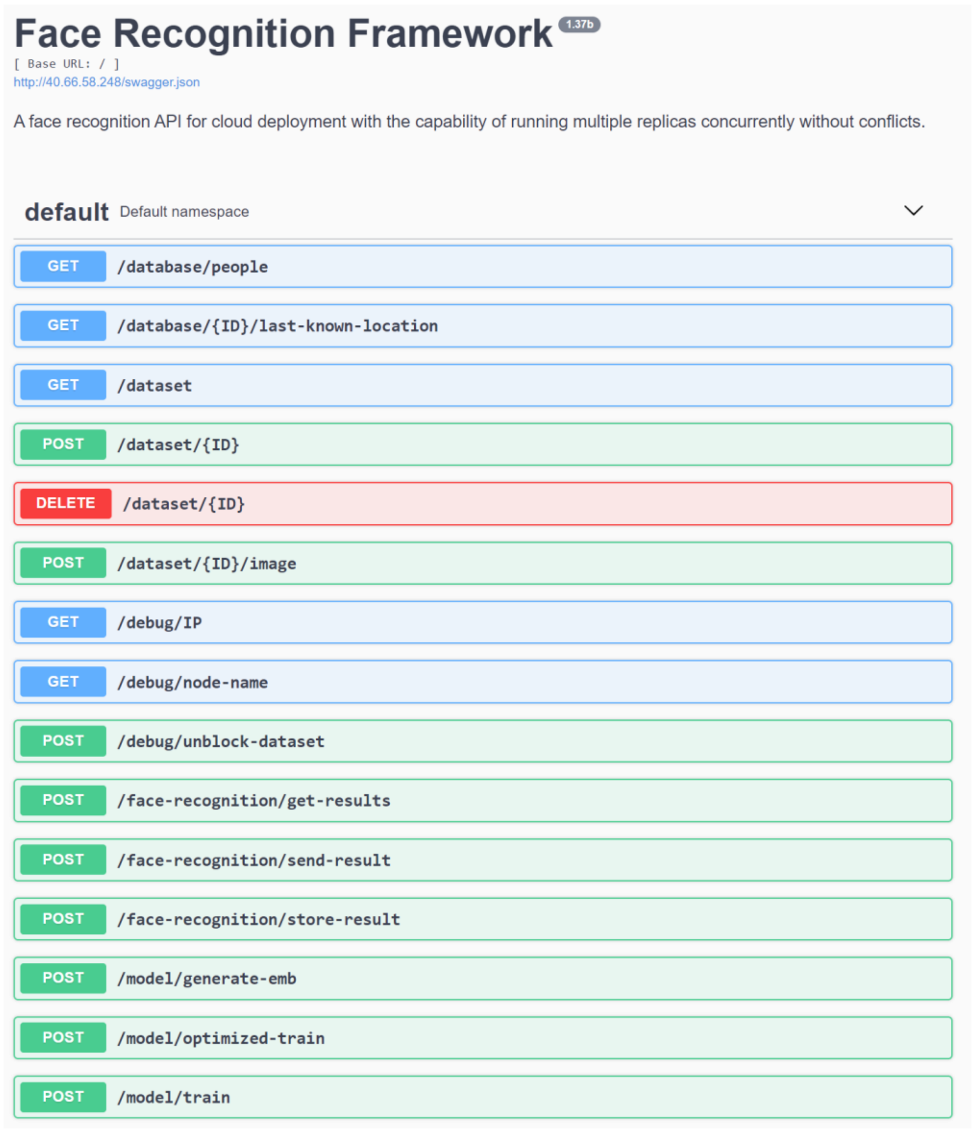
Number of Requests	Seconds to Complete	Requests per Second
100	15.156	6.846
200	21.604	9.278
300	27.052	11.112

Table 4: Relation Between Number of Requests and Requests per Second for a Ten Pod Deployment

6.2 Online Swagger Documentation

The Cloud API includes an online documentation that can be used as a reference manual for all the API calls, as well as a platform to send one-time queries, debugging queries or manually perform any function of the system.

This documentation, as shown in Figure 46, includes a description of all the methods available in the Cloud API, with their addresses and the types of HTTP Requests to be performed.



Face Recognition Framework 1.37b

[Base URL: /]
http://40.66.58.248/swagger.json

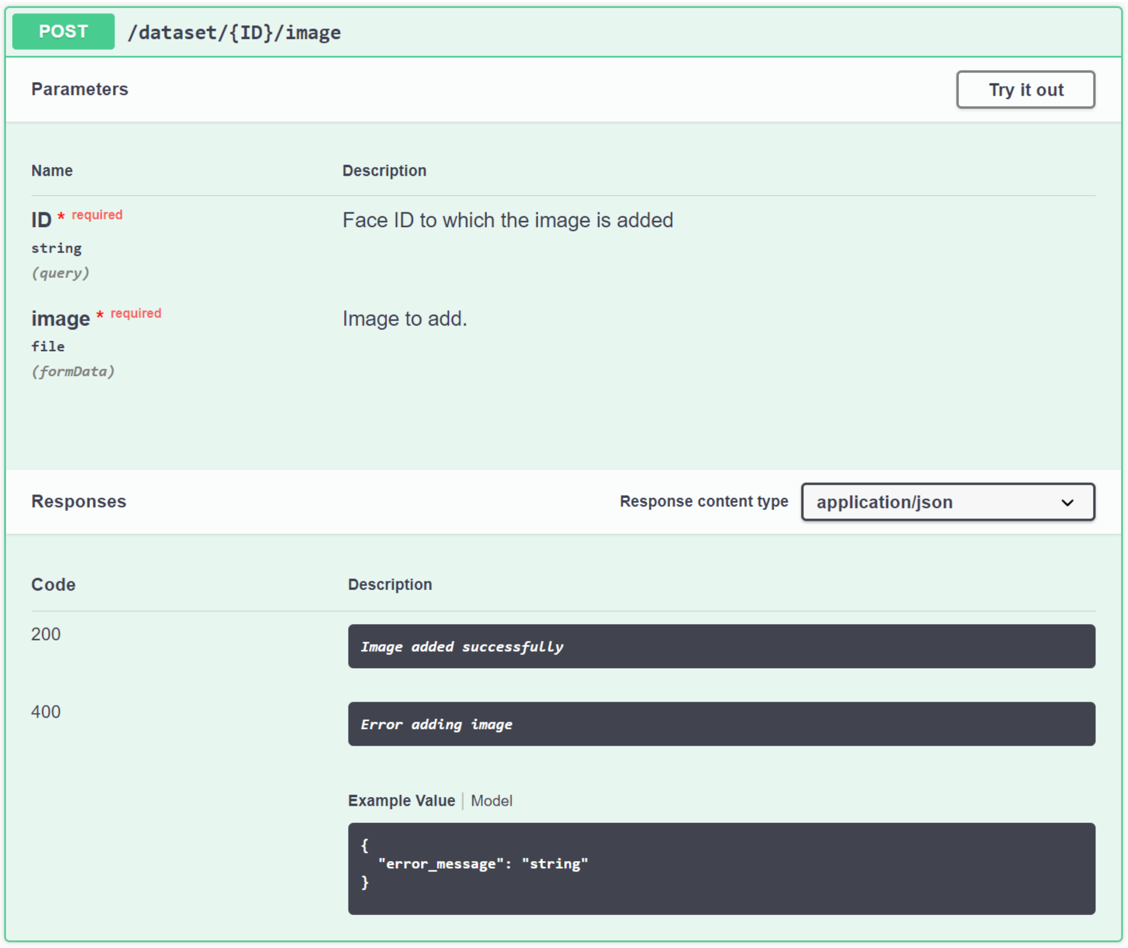
A face recognition API for cloud deployment with the capability of running multiple replicas concurrently without conflicts.

default Default namespace

```
GET     /database/people
GET     /database/{ID}/last-known-location
GET     /dataset
POST    /dataset/{ID}
DELETE  /dataset/{ID}
POST    /dataset/{ID}/image
GET     /debug/IP
GET     /debug/node-name
POST    /debug/unblock-dataset
POST    /face-recognition/get-results
POST    /face-recognition/send-result
POST    /face-recognition/store-result
POST    /model/generate-emb
POST    /model/optimized-train
POST    /model/train
```

Figure 46: Swagger Online Cloud API Documentation

If one of the views of the API calls is expanded as shown in Figure 47, the manual expands on the input data to the call, including data types, an example result of a successful and unsuccessful execution of the call, and the meaning of the different HTTP status codes. It also provides the option to perform an API call via the “Try it out” button.



/dataset/{ID}/image

Parameters

Name	Description
ID * required string (query)	Face ID to which the image is added
image * required file (formData)	Image to add.

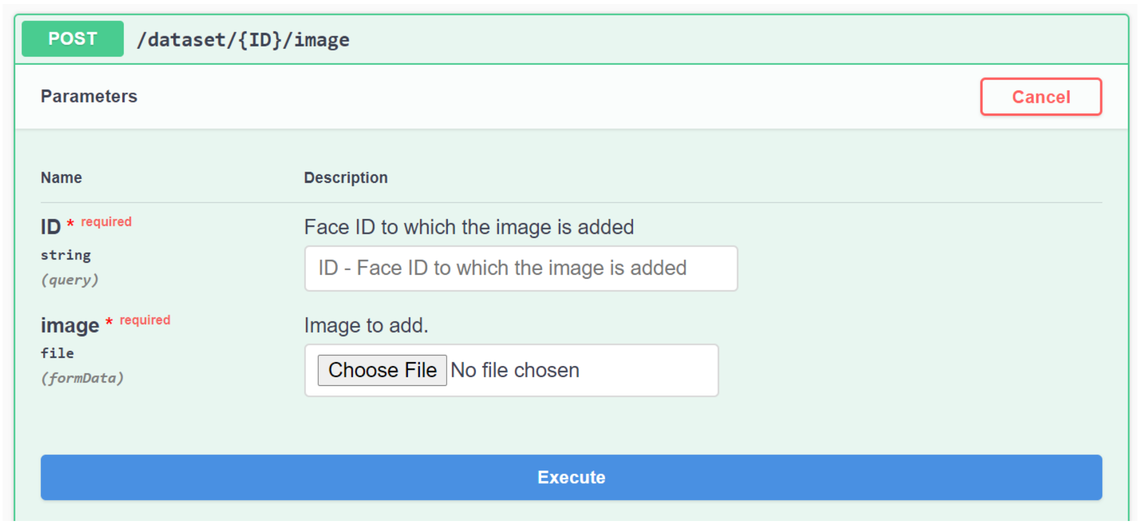
Responses

Code	Description
200	Image added successfully
400	Error adding image

```
{
  "error_message": "string"
}
```

Figure 47: Expanded View of the Add Image Call of the Online Cloud API Documentation

By clicking the “try it out” button in Figure 47, an interface like the one shown in Figure 48 is offered. From this interface, the user can perform the API call from the browser itself, which simplifies testing and allows quick interaction and learning of the framework.


POST /dataset/{ID}/image

Parameters

Name	Description
ID * required string (query)	Face ID to which the image is added ID - Face ID to which the image is added
image * required file (formData)	Image to add. Choose File No file chosen

Figure 48: "Try it out" View of the HTTP POST /dataset/<ID>/image Call of the Cloud API Documentation

Once the function is executed, it also shows the result obtained and the data returned by the API (in case there is any). With some of the functions (as the one in this example), the only data returned by the API is an HTTP code 200, which means that the request has been processed successfully and there is no further work to be done.

6.3 Add ID and Images from Dataset

A program has been developed to add new users to the dataset. This program takes instructions from the Command Line Interface (CLI) and obtains the images from the folder described in the instructions.

The program can add Face IDs, add images to the system for a given Face ID and instruct the Cloud API to perform an optimized training based on the labels and embeddings files in the Dataset folder.

The input arguments to the program are:

- Input-folder: Points to the address of the directory with the images to be uploaded.
- Person-ID: Name of the Face ID to which the images are to be added.
- IP-address: The Cloud API ingress IP address (points to the Load Balancer).
- Create-ID: If present in the command, indicates that the program must sent a Face ID creation request prior to sending the images.
- Train: If present in the command, indicates that the program must send an Optimized train request to the Cloud API once the ID has been created and the images have been added.

As shown in the flowchart in Figure 49, the program first checks whether it has to add the face ID and, if so, it sends a request to the Cloud API. Afterwards, the program adds all the images in the input-folder argument. Finally, if the train argument is present in the command, the program sends an optimized train request to the Cloud API.

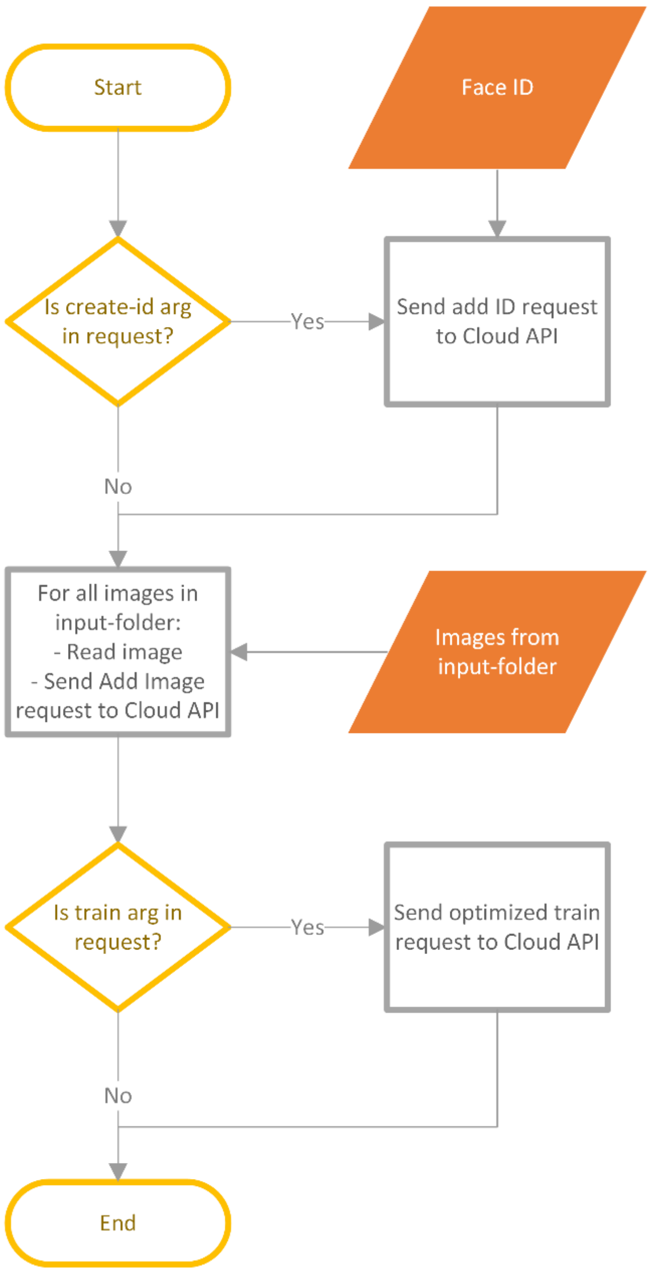

Start
Face ID
Is create-id arg in request?
Yes
Send add ID request to Cloud API
No
For all images in input-folder:
- Read image
- Send Add Image request to Cloud API
Images from input-folder
Is train arg in request?
Yes
Send optimized train request to Cloud API
No
End

Figure 49: Add ID and Images from Dataset Program Flowchart

In this program, the images are not processed locally by the program, as they are supposed to be preprocessed and to only contain the images of the subject to be added to the system.

An example of the command used to run this program and the console output during execution can be found in Figure 50. The HTTP response code 200 indicates that there were no problems during in the execution of the request.

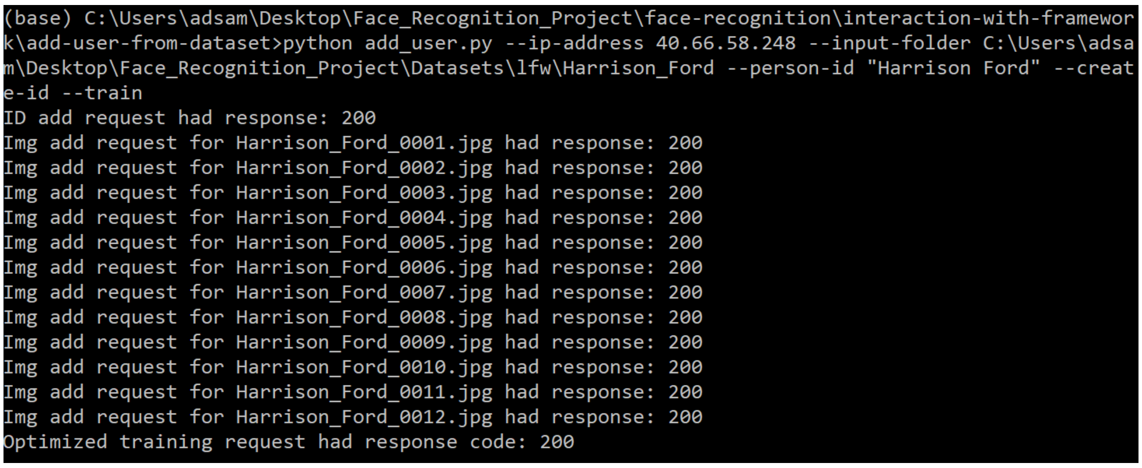

```
(base) C:\Users\adsam\Desktop\Face_Recognition_Project\face-recognition\interaction-with-framewor
k\add-user-from-dataset>python add_user.py --ip-address 40.66.58.248 --input-folder C:\Users\adsa
m\Desktop\Face_Recognition_Project\Datasets\lfw\Harrison_Ford --person-id "Harrison Ford" --creat
e-id --train
ID add request had response: 200
Img add request for Harrison_Ford_0001.jpg had response: 200
Img add request for Harrison_Ford_0002.jpg had response: 200
Img add request for Harrison_Ford_0003.jpg had response: 200
Img add request for Harrison_Ford_0004.jpg had response: 200
Img add request for Harrison_Ford_0005.jpg had response: 200
Img add request for Harrison_Ford_0006.jpg had response: 200
Img add request for Harrison_Ford_0007.jpg had response: 200
Img add request for Harrison_Ford_0008.jpg had response: 200
Img add request for Harrison_Ford_0009.jpg had response: 200
Img add request for Harrison_Ford_0010.jpg had response: 200
Img add request for Harrison_Ford_0011.jpg had response: 200
Img add request for Harrison_Ford_0012.jpg had response: 200
Optimized training request had response code: 200
```

Figure 50: Console Output for the Add ID and Images from Dataset CLI Program

The time it takes the system from the start of the program execution until the new ID is added, the images are added, the system has been trained to recognize the new person (and any new recognition request will use the newly trained classifier) can be as low as 30 seconds or less (in the example in Figure 50 with 12 images it was 26 seconds). The specific time will depend on the number of images to be added and the number of images already present in the dataset, as the SVM classifier will take longer to train for higher number of dataset images (larger Labels and Embeddings files), even when using the optimized train function (which omits the generation of the image embeddings).

As previously mentioned, this process is completely compatible with the recognition of images and allows the system to keep operational while the ID and images are added and while the SVM classifier is trained. Once this process is finished, the system will start using the newly trained classifier seamlessly and without any interruption to system operations.

6.4 Add ID and Images from Camera

The Add ID and Images from Camera program aids with the task of adding a new user to the face recognition system. The difference between this program and the Add ID and Images from Dataset one lies on the source of the images added to the Face ID. Instead of using pre-processed images available in a folder, this program uses the webcam connected to the computer where the program is running to capture the images which it then processes by extracting the faces to later send them to the Cloud API.

At the end of the process, the program can also instruct the Cloud API to perform an optimized training so that the new face is recognized using the face recognition API calls.

The input arguments of the program are:

- Person-ID: Name of the Face ID to which the images are to be added.
- IP-address: The Cloud API ingress IP address (the address points to the Load Balancer).
- Img-number: the number of face images to be taken, pre-processed and sent to the Cloud API to add them to the Face ID.
- Create-ID: If present in the command, indicates that the program must sent a Face ID creation request prior to sending the images.

- Train: If present in the command, indicates that the program must send an Optimized train request to the cloud API once the ID has been created and the images have been added.

As shown in the flowchart in Figure 51, the program first loads the tensorflow model used to detect the faces. If the create-id argument is present, it sends the add ID request to the Cloud API. Afterwards, the program captures images from the camera, extracts the faces in the images and sends these images to the Cloud API. Once the number of face images sent to the cloud matches the number given in the img-number argument, the camera is released. Finally, if the train argument is provided, an optimized train request is sent to the Cloud API.

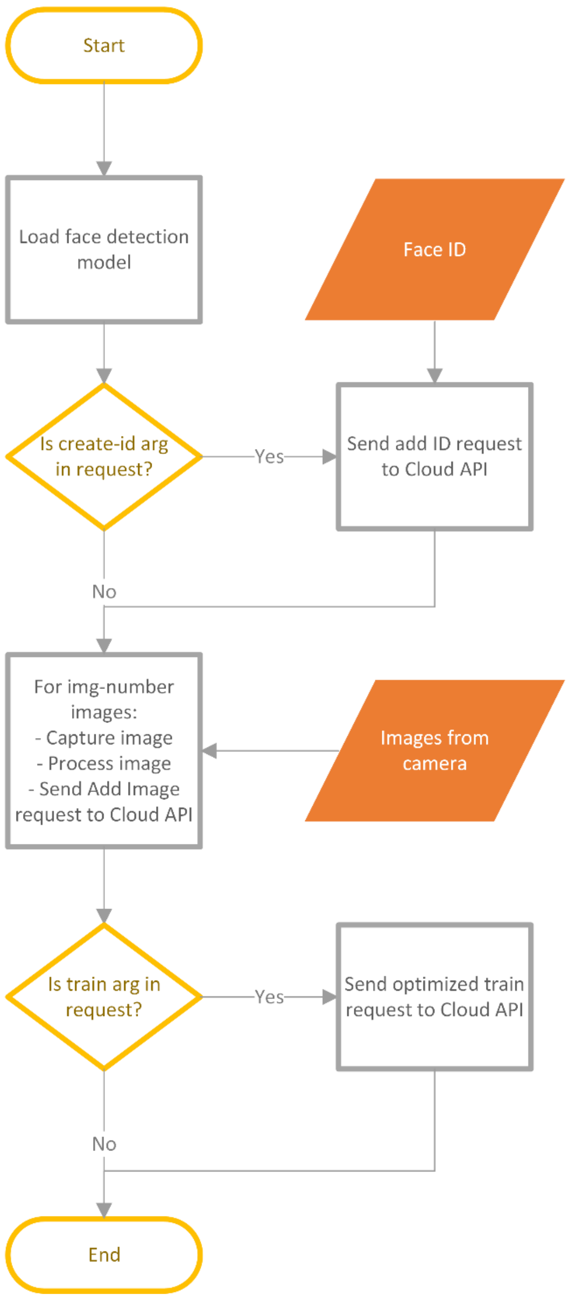
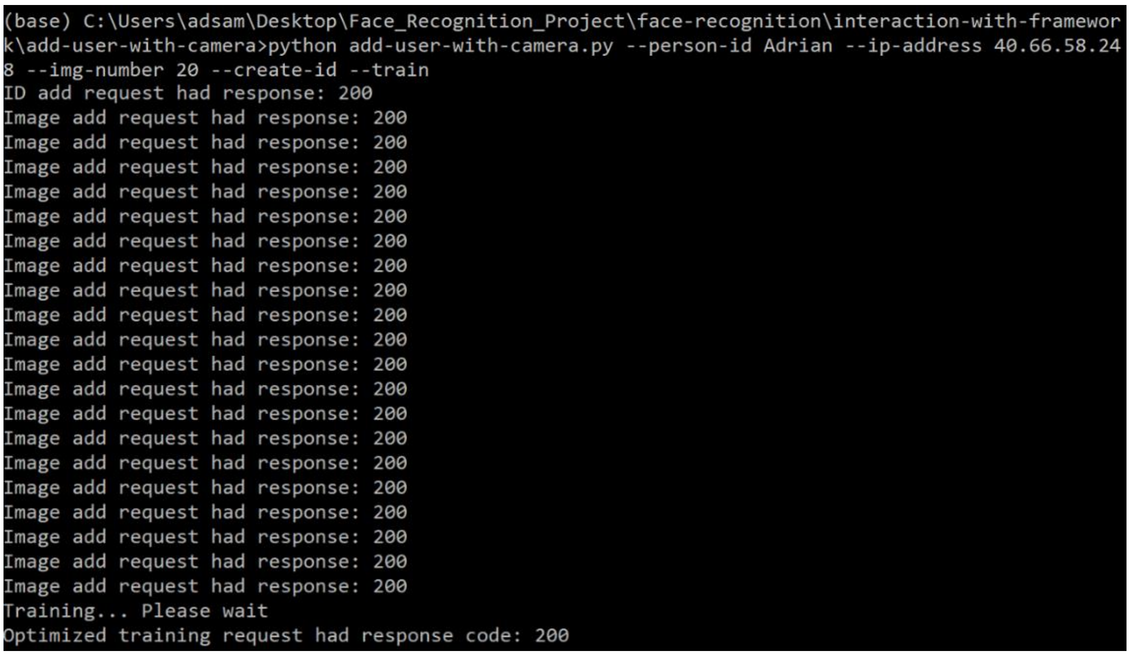

Start
Load face detection model
Face ID
Is create-id arg in request?
Yes
Send add ID request to Cloud API
No
For img-number images:
- Capture image
- Process image
- Send Add Image request to Cloud API
Images from camera
Is train arg in request?
Yes
Send optimized train request to Cloud API
No
End

Figure 51: Add ID and Images from Camera Program Flowchart

Note that if more than one face is detected in the images from the camera, the system will add none for that frame, as it does not know which one is the user that is being added. Once only one face is detected, the system will resume sending the face images to the cloud until the img-number is reached.

An example of the command used to run this program and part of the output provided by the program in the CLI is shown in Figure 52.As in the other programs, the code 200 means that the request has been processed successfully.



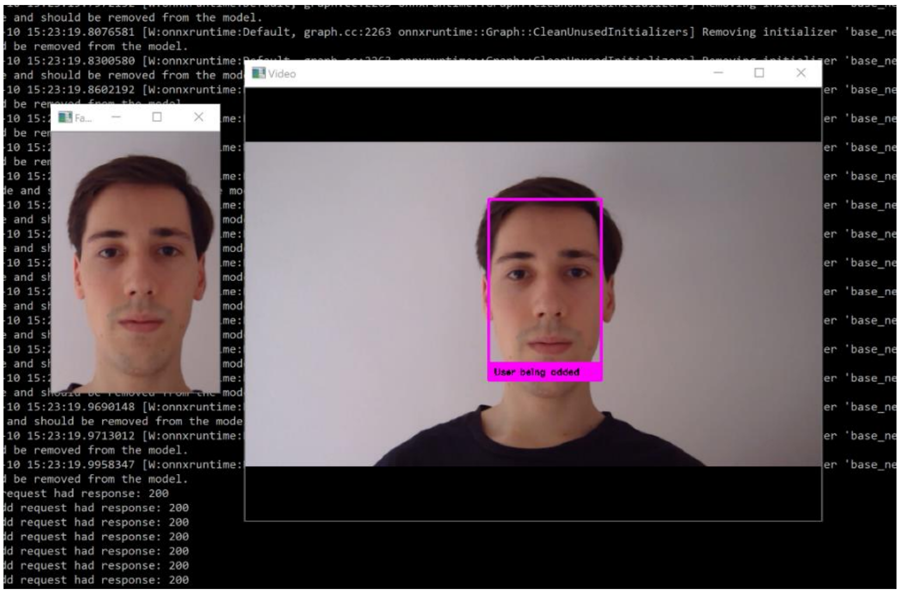
```
(base) C:\Users\adsam\Desktop\Face_Recognition_Project\face-recognition\interaction-with-framewor
k\add-user-with-camera>python add-user-with-camera.py --person-id Adrian --ip-address 40.66.58.24
8 --img-number 20 --create-id --train
ID add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Image add request had response: 200
Training... Please wait
Optimized training request had response code: 200
```

Figure 52: Console Output for the Add ID and Images from Camera CLI Program

The process to add the images from the camera takes more time than if the picture is feed from a folder, as the face picture has to be extracted and sent to the Cloud API, and these capture and extraction processes take more time.

The face detection and extraction processes are performed so that the images sent to the Cloud API do contain a face. Given the extra processing involved, the total time to add 20 images of a subject shown in Figure 52 took approximately one minute to complete.

The program displays one window with the images captured by the camera with a purple box surrounding the face being added and another window with the face image being sent to the Cloud API. An example of the contents of the general view window is shown in Figure 53.



```
e and should be removed from the model.
-10 15:23:19.8076581 [W:onnxruntime:Default, graph.cc:2263 onnxruntime::Graph::CleanUnusedInitializers] Removing initializer 'base_ne
d be removed from the model.
-10 15:23:19.8300580 [W:onnxruntime:                                                                                      er 'base_ne
e and should be removed from the mod                                                                                      er 'base_ne
-10 15:23:19.8602192 [W:onnxruntime:                                                                                      er 'base_ne
-10 15:23:19.9690148 [W:onnxruntime:                                                                                      er 'base_ne
 and should be removed from the mode                                                                                      er 'base_ne
-10 15:23:19.9713012 [W:onnxruntime:                                                                                      er 'base_ne
d be removed from the model.
-10 15:23:19.9958347 [W:onnxruntime:
d be removed from the model.
request had response: 200
dd request had response: 200
dd request had response: 200
dd request had response: 200
dd request had response: 200
dd request had response: 200
dd request had response: 200
```

Figure 53: Windows for the Add ID and Images from Camera Program

The actual image being sent to the Cloud API is slightly larger than the box in Figure 53. This is important, as a tightly cut face image may not perform well in the MTCNN face detection algorithm on the Cloud API.

6.5 Real-Time Face Recognition from Camera

A program has been developed to allow for face recognition from the webcam in the computer running the program. It allows to capture images from the camera, detect and extract the faces, send the detected faces to the Cloud API, get the identities for the faces and display the identities as an overlay to the original image.

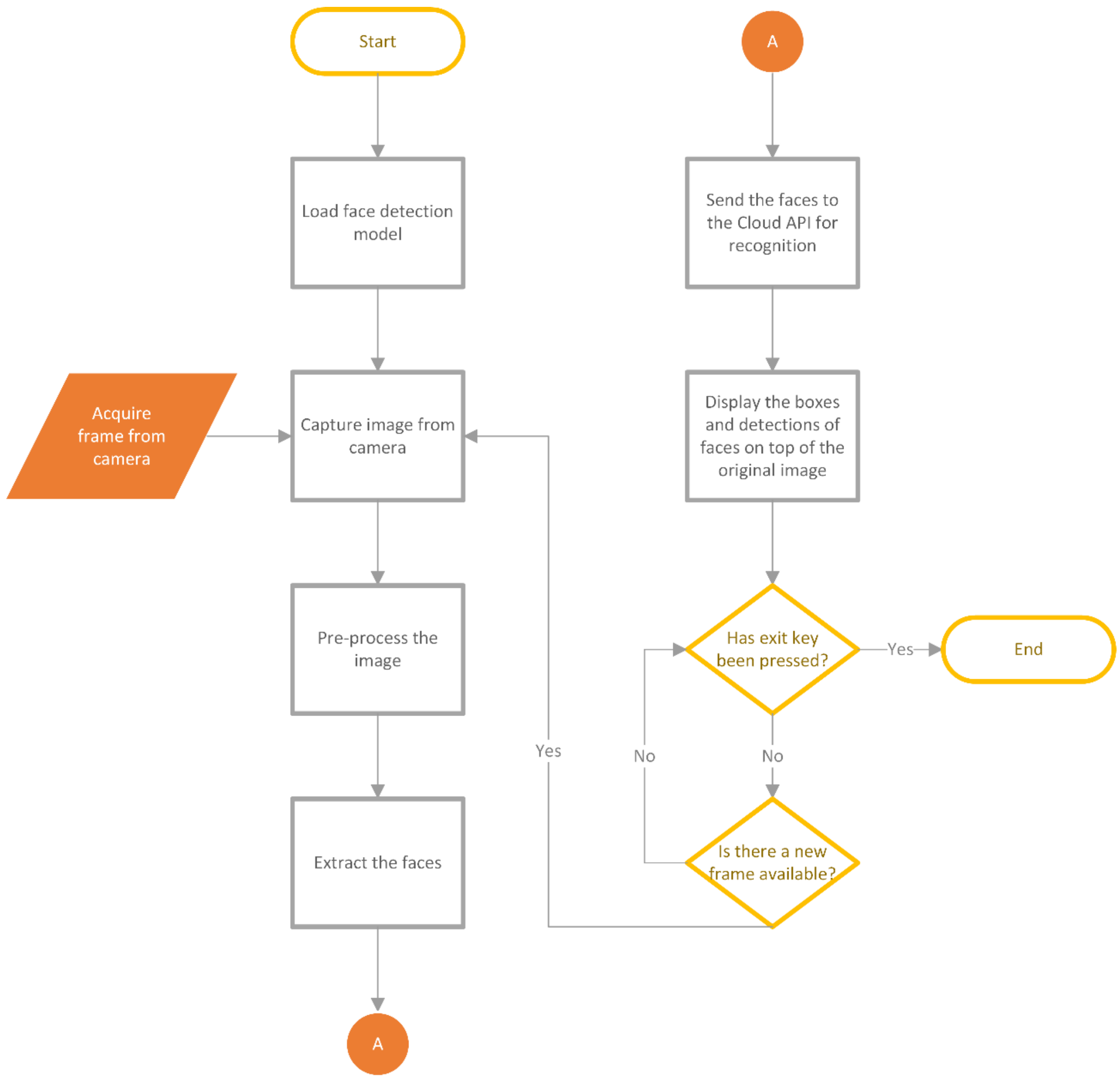
The input arguments of the program are:

- IP-address: The Cloud API ingress IP address (the address points to the Load Balancer).
- Min-size: Minimum face image size on the long axis. Smaller images are discarded.
- Max-size: Maximum face image size on the long axis. Larger images are resized.
- Min-confidence: Minimum confidence level to show result of face recognition.
- Multithreading: Send all face images in a frame simultaneously to the Cloud API (improves performance).

The box surrounding the faces is color-coded to represent the result of the face recognition:

- No box: the face is not detected by the local program.
- Red box: the face candidate is detected by the local program, although the Cloud API does not have enough confidence that the image sent contains a face to recognize it.
- Blue box: the face is detected by both the local program and the Cloud API, but the Cloud API does not have enough confidence about the identity of the face so as to show it in the box (this confidence level can be changed in the local program).
- Green box with recognized identity: the face is detected by both the local program and the Cloud API, and the face is recognized by the Cloud API with enough confidence to show it in the box label.

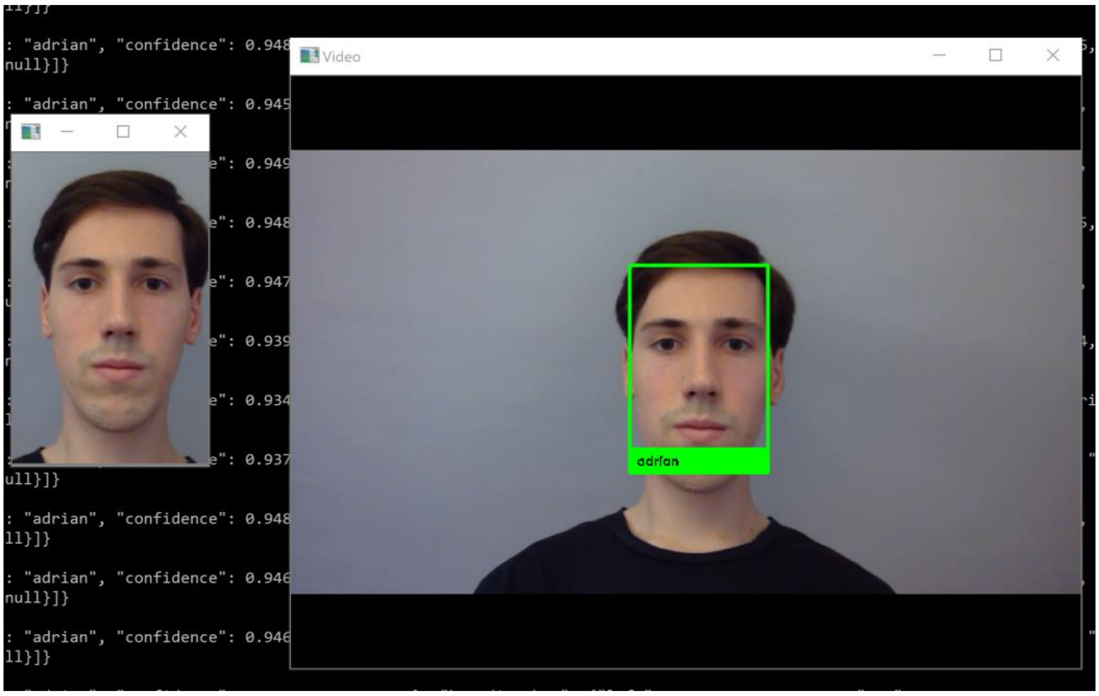
The face recognition of the images from the camera is achieved by following the procedure in Figure 54. The program starts by loading the face detection model. It then obtains an image from the camera, processes it, extracts the faces, and sends them to the Cloud API for face recognition. With the results of the face recognition, the program displays the boxes surrounding the faces in the original frame, with the detected identities.



Start
Load face detection model
Acquire frame from camera
Capture image from camera
Pre-process the image
Extract the faces
A
A
Send the faces to the Cloud API for recognition
Display the boxes and detections of faces on top of the original image
Has exit key been pressed?
Yes
End
No
No
Is there a new frame available?
Yes

Figure 54: Real-Time Face Recognition from Camera Program Flowchart

The program provides the two windows shown in Figure 55: one with the original image captured by the camera and another one with a slightly larger crop of the face image being sent to the Cloud API.



```
: "adrian", "confidence": 0.948
null}]}
: "adrian", "confidence": 0.945
e": 0.949
e": 0.948
e": 0.947
e": 0.939
e": 0.934
e": 0.937
ull}]}
: "adrian", "confidence": 0.948
ll}]}
: "adrian", "confidence": 0.946
null}]}
: "adrian", "confidence": 0.946
ll}]}
```

Figure 55: Camera Windows from the Real-Time Face Recognition from Camera Program

Sending to the cloud an image covering more area than the actual face detected is important, as a tightly cut face image may not perform well in the MTCNN face detection algorithm on the Cloud API.

Note that all the programs are also capable of detecting multiple faces in the same frame. In the case of multiple faces present in the frame, the face images are alternated in the cropped view window on the left in Figure 55.

6.6 Face Recognition from Video File or Video Stream

To perform the face recognition and the display of results in the images from a video file or a video stream, the video recognition program allows to select the source of the video and then performs the recognition in this video.

The input arguments of the program are:

- IP-address: The Cloud API ingress IP address (the address points to the Load Balancer).
- Video-source: The web address of the video stream or the file path of the video file.
- Skip-frames: Number of frames to be skipped. For 30fps, if set to 5, the effective framerate is 5fps.
- Target-fps: Approximate target frames per second. Only for video files.
- Min-size: Minimum face image size on the long axis. Smaller images are discarded.
- Max-size: Maximum face image size on the long axis. Larger images are resized.
- Min-confidence: Minimum confidence level to show result of face recognition.
- Multithreading: Send all face images in a frame simultaneously to the Cloud API (improves performance).

The procedure followed by the program in order to achieve the recognition and to display the results is shown in Figure 56. This procedure is the same as the one of the real-time camera recognition but changing the source of the images from the integrated camera to the video file or video stream of choice. It is also worth to mention that all the frames from a file are processed, while in the case of the camera, if there are significant processing delays, many frames could be skipped.

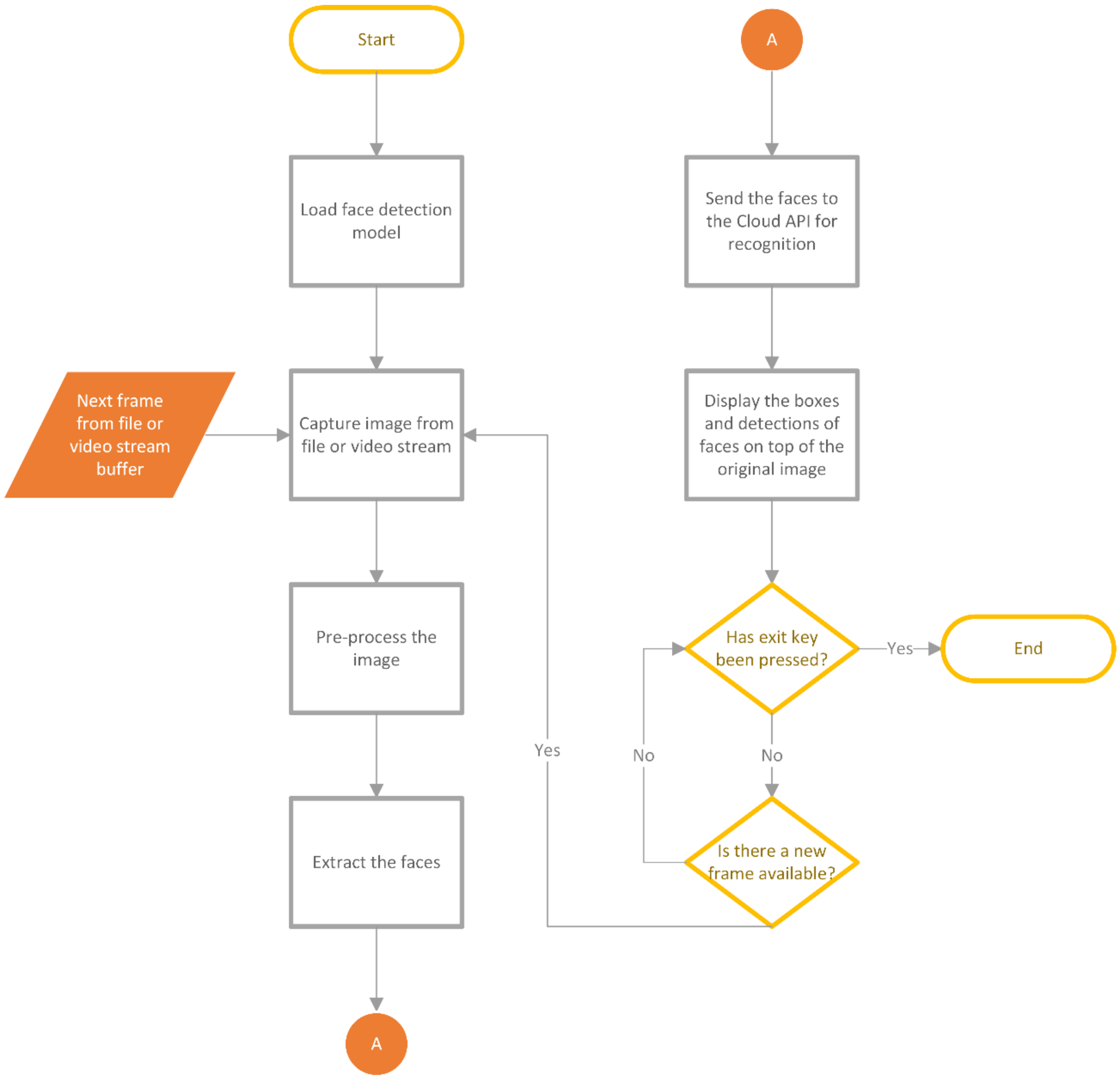
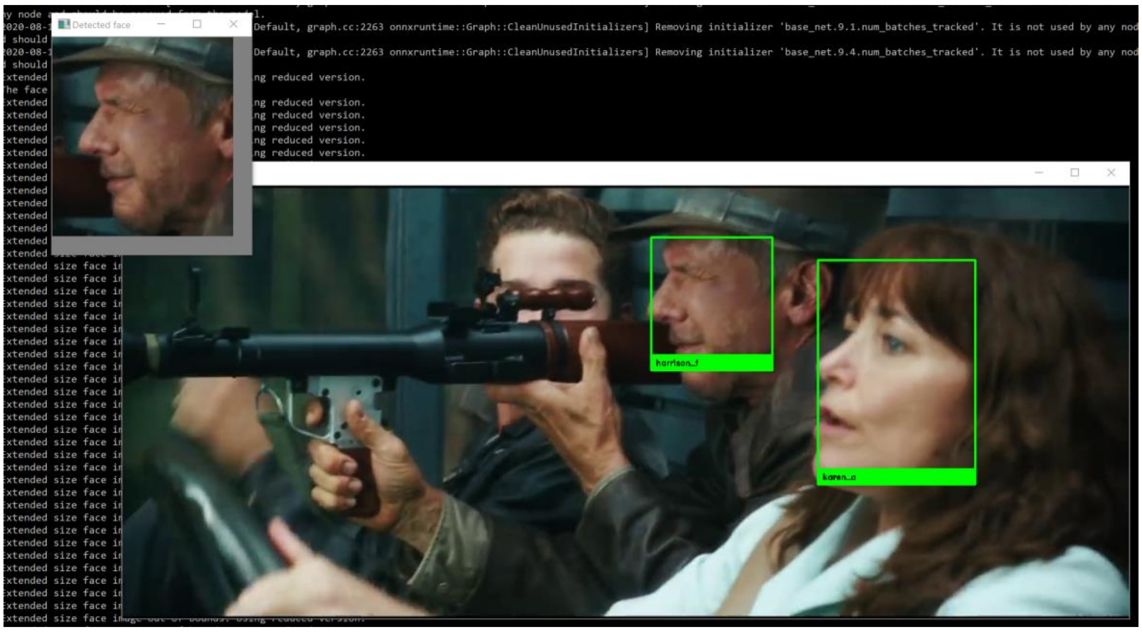

Start
Load face detection model
Next frame from file or video stream buffer
Capture image from file or video stream
Pre-process the image
Extract the faces
A
A
Send the faces to the Cloud API for recognition
Display the boxes and detections of faces on top of the original image
Has exit key been pressed?
Yes
End
No
Is there a new frame available?
No
Yes

Figure 56: Face Recognition from Video File or Video Stream Program Flowchart

The user interface of this program is based on the interface in the real-time face recognition with camera program. The faces are either not detected (no bounding box), detected by the local program but not by the Cloud API (red box), detected by both the local program and the Cloud API but not recognized (blue box), or detected by both the local program and the cloud API and recognized (green box with the recognized identity).

The results of this program are displayed similarly to those of the program that recognizes the faces from the video recorded by the camera. As shown in Figure 57, the data is divided in two windows: one for the video frames with the overlay of the boxes and another for the face images being sent to the cloud for recognition.



```
y node a                                           l.
2020-08-1                                          Default, graph.cc:2263 onnxruntime::Graph::CleanUnusedInitializers] Removing initializer 'base_net.9.1.num_batches_tracked'. It is not used by any nod
d should
2020-08-1                                          Default, graph.cc:2263 onnxruntime::Graph::CleanUnusedInitializers] Removing initializer 'base_net.9.4.num_batches_tracked'. It is not used by any nod
d should
xtended                                            ng reduced version.
The face
xtended                                            ng reduced version.
xtended                                            ng reduced version.
xtended                                            ng reduced version.
xtended                                            ng reduced version.
xtended                                            ng reduced version.
xtended
xtended
xtended
xtended
xtended
xtended
xtended
xtended
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face in
xtended size face image out of bounds. Using reduced version.
```

Figure 57: Windows from the Face Recognition from Video Program [53]

An example of how the frame window looks when fed a frame with known faces (green), unknown faces (blue), and bad quality or obstructed face images (red) is shown in Figure 58.

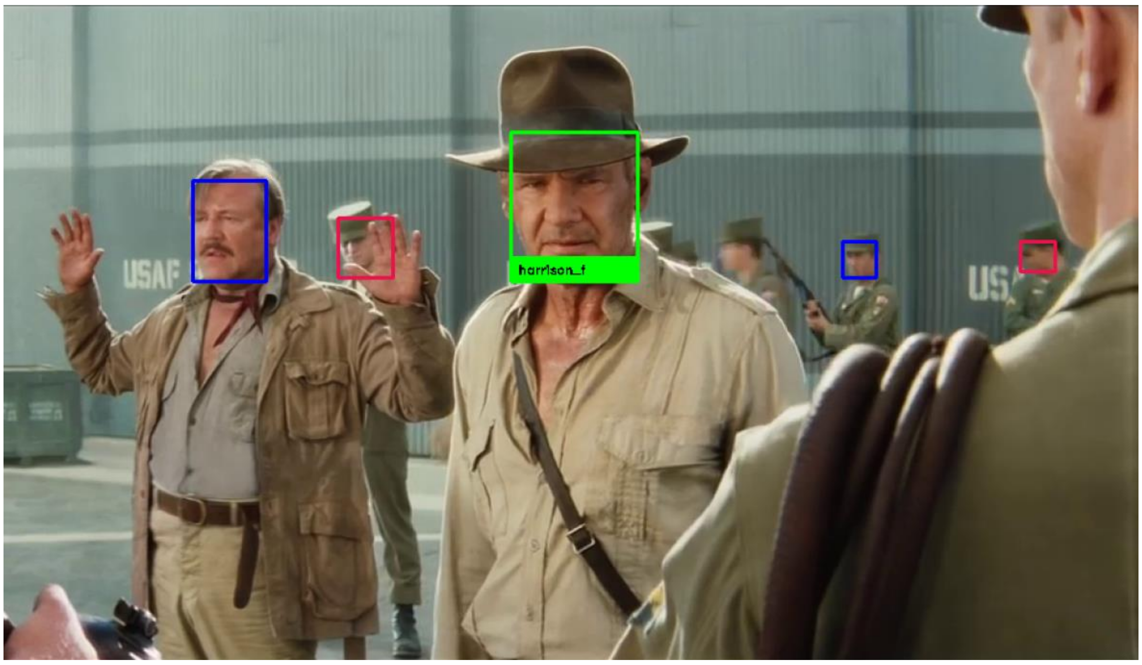


Figure 58: Example Scene for the Face Recognition from Video Program [53]

6.7 Face Extraction from Video File or Video Stream

A program has been designed to extract the faces in a video file or stream. It is used to test the bandwidth savings achieved by using the architecture proposed in this project.

The input arguments of the program are:

- Video-source: The web address of the video stream or the file path of the video file.
- Output-folder: Output folder where the face images extracted are stored.

- Skip-frames: Number of frames to be skipped. For 30 fps, if set to 5, the effective framerate is 5 fps.
- Target-fps: Approximate target frames per second. Only for video files.
- Min-size: Minimum face image size on the long axis. Smaller images are discarded.
- Max-size: Maximum face image size on the long axis. Larger images are resized.

The procedure followed by the program to extract the faces from the video stream or file and to store them to a folder is shown in Figure 59. The extraction process is similar to that performed in the video recognition, storing the face images to a local folder instead of sending the images to the Cloud API.

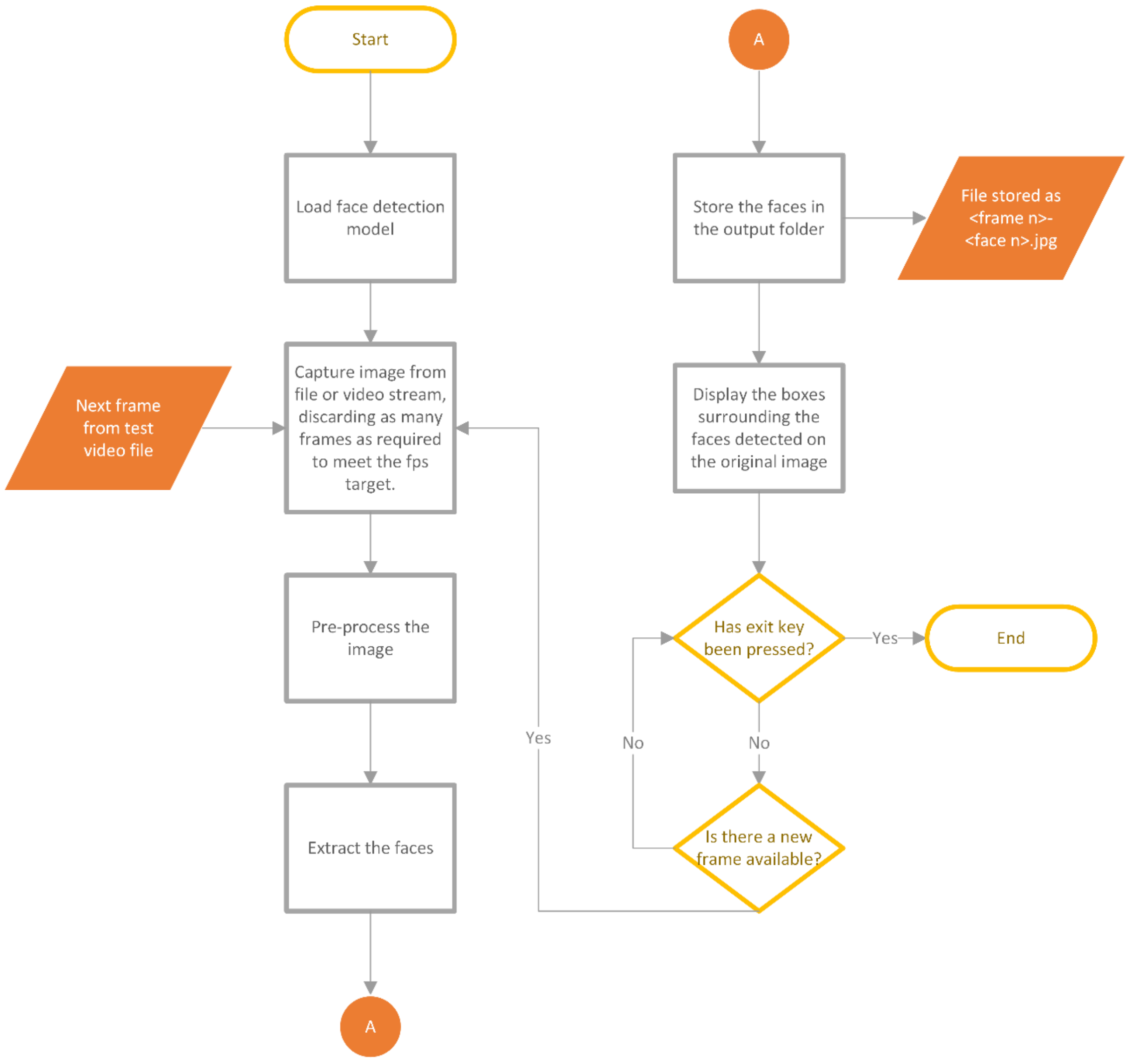

Start
A
Load face detection model
Store the faces in the output folder
File stored as <frame n>-<face n>.jpg
Next frame from test video file
Capture image from file or video stream, discarding as many frames as required to meet the fps target.
Display the boxes surrounding the faces detected on the original image
Pre-process the image
Has exit key been pressed?
Yes
End
Yes
No
No
Extract the faces
Is there a new frame available?
A

Figure 59: Face Extraction and Storage from Video File Program Flowchart

The user interface of this program is quite similar to the one of the face recognition from video file or video stream, with a different color coding and without the names of the recognized faces, as this program does not send the images to the Cloud API, neither it recognizes them locally. The only color used, as shown in Figure 60, is white, meaning that the face has been detected by the local program and is going to be stored into the local folder selected (if the face image size on the long axis is greater than the fixed minimum.

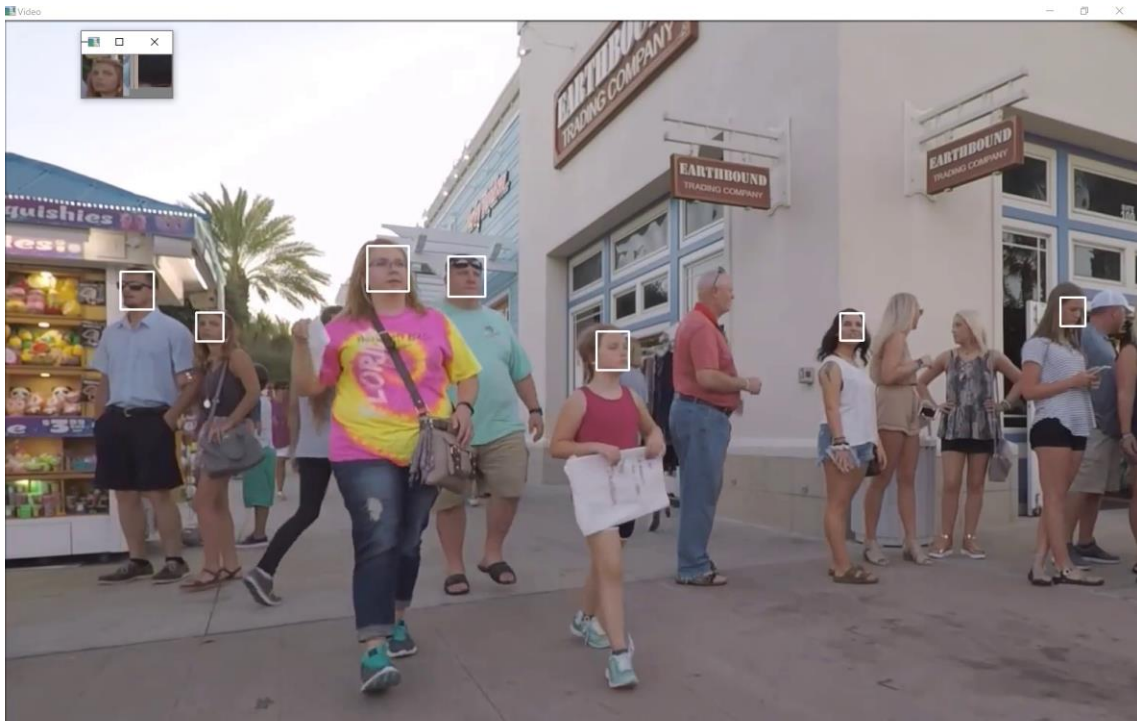

Video
EARTHBOUND TRADING COMPANY
EARTHBOUND TRADING COMPANY
EARTHBOUND TRADING COMPANY

Figure 60: Frame and Extracted Face Image Windows Obtained with the Face Extraction from Video Program

6.8 Face Recognition from Folder

The face recognition from folder program provides an interface to send the face images from a folder to the Cloud API for recognition. The input arguments of the program are:

- IP-address: The Cloud API ingress IP address (the address points to the Load Balancer).
- Input-folder: The folder with the images to upload.
- Dry-run: If included, the images are sent to the cloud but no results are obtained. For benchmarking purposes.
- Stats: Shows statistics of the images sent.

The procedure followed by the program to send the images from the folder to the Cloud API for recognition is shown in Figure 61. The program reads all the images from the folder and sends them to the Cloud API. If the dry-run parameter is present, then the images are sent to a special dummy address that will not send back any results. This address is used for benchmarking purposes.

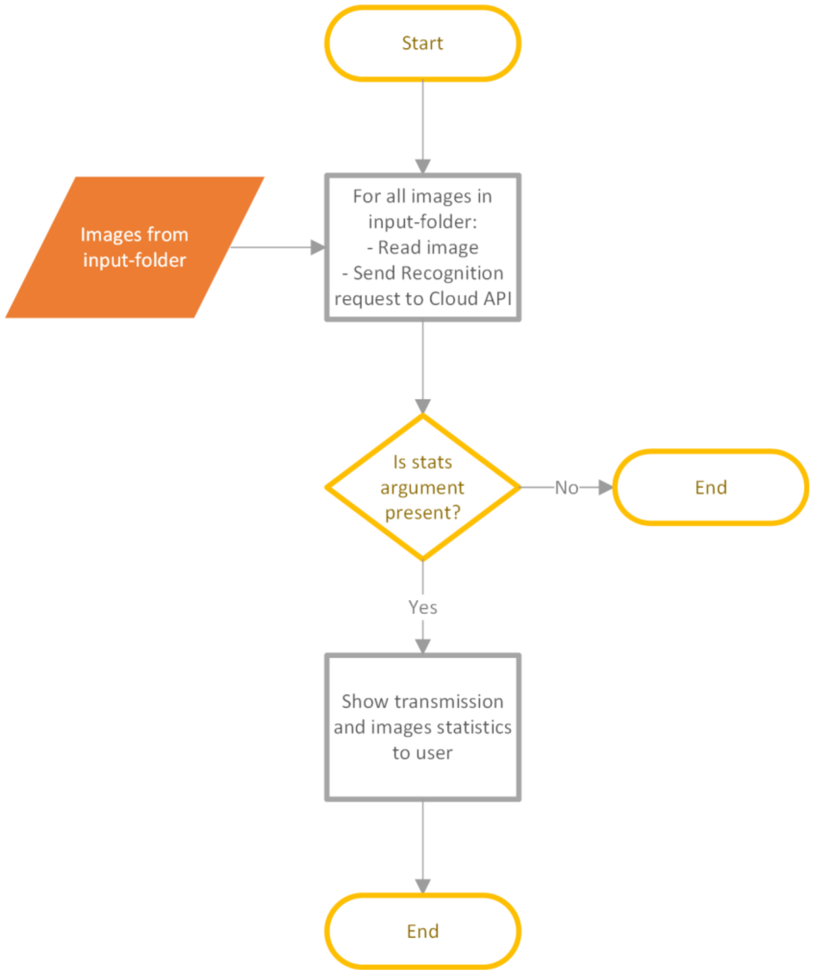
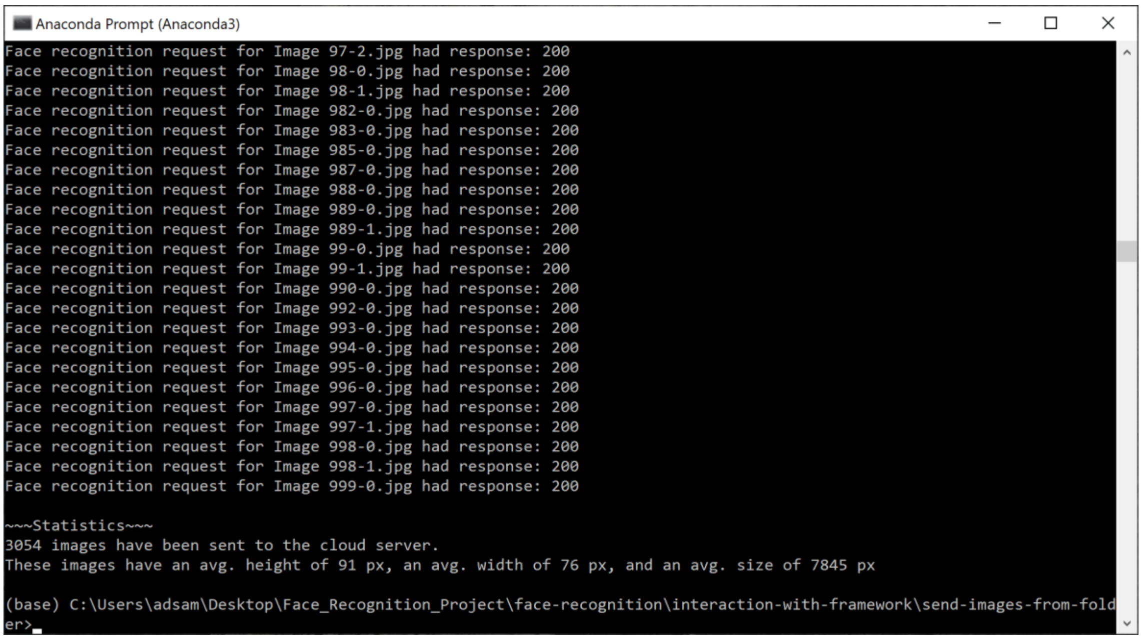

Start
Images from input-folder
For all images in input-folder:
- Read image
- Send Recognition request to Cloud API
Is stats argument present?
No
End
Yes
Show transmission and images statistics to user
End

Figure 61: Face Recognition from Folder Program Flowchart

This program provides all the data through the CLI, as shown in Figure 62. Therefore, there is only one window open with the output of the program. It provides confirmation of the submission of the images and, in case the dry-run argument is not present, it also provides the recognition results. At the end of the execution, if requested, it provides statistics regarding the submission.



```
Anaconda Prompt (Anaconda3)
Face recognition request for Image 97-2.jpg had response: 200
Face recognition request for Image 98-0.jpg had response: 200
Face recognition request for Image 98-1.jpg had response: 200
Face recognition request for Image 982-0.jpg had response: 200
Face recognition request for Image 983-0.jpg had response: 200
Face recognition request for Image 985-0.jpg had response: 200
Face recognition request for Image 987-0.jpg had response: 200
Face recognition request for Image 988-0.jpg had response: 200
Face recognition request for Image 989-0.jpg had response: 200
Face recognition request for Image 989-1.jpg had response: 200
Face recognition request for Image 99-0.jpg had response: 200
Face recognition request for Image 99-1.jpg had response: 200
Face recognition request for Image 990-0.jpg had response: 200
Face recognition request for Image 992-0.jpg had response: 200
Face recognition request for Image 993-0.jpg had response: 200
Face recognition request for Image 994-0.jpg had response: 200
Face recognition request for Image 995-0.jpg had response: 200
Face recognition request for Image 996-0.jpg had response: 200
Face recognition request for Image 997-0.jpg had response: 200
Face recognition request for Image 997-1.jpg had response: 200
Face recognition request for Image 998-0.jpg had response: 200
Face recognition request for Image 998-1.jpg had response: 200
Face recognition request for Image 999-0.jpg had response: 200

~~~Statistics~~~
3054 images have been sent to the cloud server.
These images have an avg. height of 91 px, an avg. width of 76 px, and an avg. size of 7845 px

(base) C:\Users\adsam\Desktop\Face_Recognition_Project\face-recognition\interaction-with-framework\send-images-from-fold
er>_
```

Figure 62: Console Output for the Face Recognition from Folder Program

6.9 Bandwidth Savings Benchmark

In these tests, the face extraction and image transmission programs are used to obtain and send the images of the faces found in the different test videos used for the experiment. The process is monitored, measuring many parameters to achieve a greater knowledge of the bandwidth-saving performance of the system.

The videos used for the tests contain images of the flow of people in various streets in different locations around the world. Each one of the videos shows a different density of people, leading to a wide variety of test scenarios. This allows to fully demonstrate the variability of the bandwidth savings achieved with the architecture designed in this document depending on the density of people in the video feeds analyzed.

The variations in the bandwidth required depends on the number of faces in a video stream. This is due to how the system achieves the bandwidth savings, which is linearly dependent on the number of faces detected and transmitted on each of the frames processed, as well as on the size of the faces in pixels. To differentiate between the different face densities, five categories will be considered as shown in Table 5, defined by the number of face images sent per minute.

Range	**Face images per minute**
Very low density	<5
Low density	5-50
Medium density	50-500
High density	500-1000
Very high density	>1000

Table 5: Face Density Ranges

In the following subsections, videos falling under the medium density, high density and very high density categories are used to test the system. These videos have each a length of 10 minutes. Links to their web addresses can be found in each of the subsections.

To emulate a security IP camera, the videos are converted to 6 fps by the face extraction program. IP video surveillance cameras require an approximate bandwidth of 4 Mbps when sending 1080P video at reasonable qualities and framerates of 6-10 fps [54] [55].

The video file size used for the comparison is determined from the assumption of the 4 Mbps bandwidth required for a surveillance camera transmitting stream of 6 fps at 1080P.

The face extraction process is configured so that face images that are too small to be recognized (less than 30 px in the short axis, in our case) are discarded, while face images that are over a certain threshold (150 px in the long axis, in our case) are compressed to reduce the space and bandwidth required to transmit and process them.

6.9.1 Medium Density

The video from [56] has a facial density that falls into the medium density category, given that, after extracting the faces from the video, as shown in Figure 63, the average number of face images detected and extracted per minute has been determined to be 305.4. The data used to obtain this metric can be found in Table 6. The total file size of the pictures, the average per image file size, and the average height and width in pixels of the face images have been determined from the analysis of the face images extracted can also be found in Table 6.

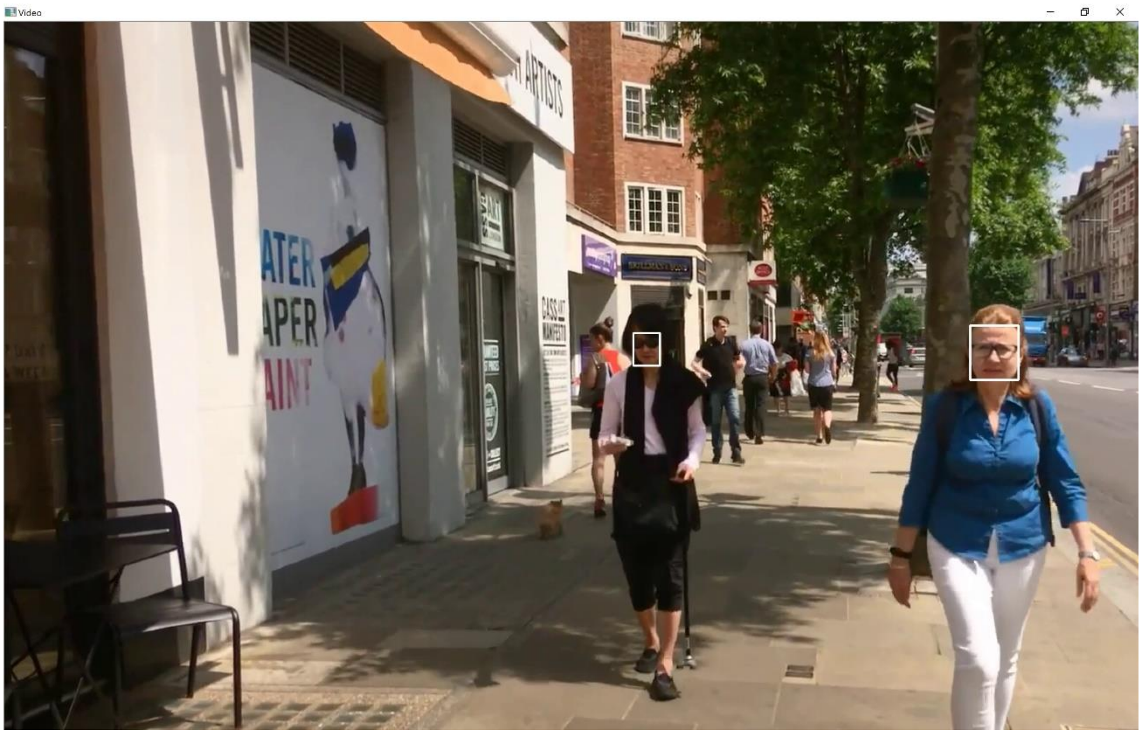


Figure 63: Face Extraction Window – Example of Medium Density

Video length	600 s
Number of face images	3054
Face images per minute	305.4
Total face image files size	10.05 MB
Average image file size	3.37 KB
Average image height	91 px
Average image width	76 px
Average image size	7845 px

Table 6: Extracted Face Images Data – Example of Medium Density

After the face images have been extracted from the video, they are sent to the cloud API. The transmission to the API has been monitored using the network capture program Wireshark [57] to obtain network usage statistics. These statistics are more adequate than the extracted image file size to determine the actual bandwidth savings obtained by using the architecture proposed in this document versus transmitting the whole video feed over the network. The results of this test are the bandwidth used by the images in the network, the average per image network size, the number of network packets sent, and the network overhead which is extra percentage used by the system when sending the images to the Cloud compared to the size of the images being sent (in this case 14 MB and 10.05 MB respectively). These results can be found in Table 7. Note that, as this test has been performed in a real environment, the transmission of the data has been affected by the network conditions, meaning that there may have been retransmission of network packets due to erroneous packets, lost packets, excessive network delays, etc. This means that if the test were repeated, the results would be quite similar but not necessarily equal.

Network packets	45536
Size in network	14 MB
Average image network size	4.69 KB
Network overhead	39.3%

Table 7: Network Transmission Data – Example of Medium Density

From the data gathered during the face image extraction and transmission, and from all the assumptions made regarding a hypothetical system sending the video camera feeds directly to the cloud (network bitrate of 4 Mbps per camera), the size reduction factor shown in Table 8 is obtained. This factor indicates a great reduction in the network bandwidth required for the recognition of the faces in the video using the proposed architecture.

Average network bitrate	0.19 Mbps
Estimated IP camera bitrate	4 Mbps
Estimated IP camera video size	300 MB
Bandwidth reduction factor (factor that divides)	21.4

Table 8: Bandwidth Reduction Factor – Example of Medium Density

6.9.2 High Density

The video from [58] has a facial density that falls into the high density category, given that after extracting the faces from the video, as shown in Figure 64, the average number of face images detected and extracted per minute has been determined to be 864.9. The data used to obtain this metric can be found in Table 9. Following the same procedure as in the medium density test, the results of the test with the high density video are also shown in Table 9**.**

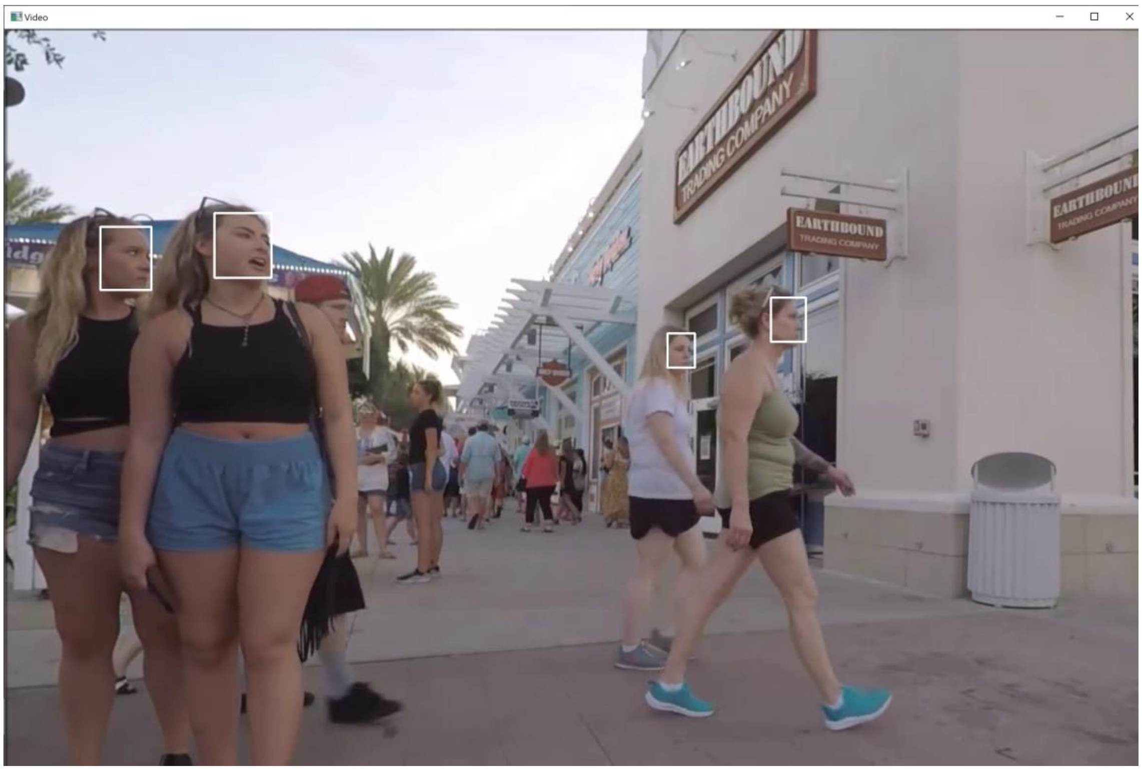


Figure 64: Face Extraction Window – Example of High Density

Video length	602 s
Number of face images	8678
Face images per minute	864.9
Total face image files size	23.8 MB
Average image file size (kb)	2.81 KB
Average image height	91 px
Average image width	76 px
Average image size (pixels)	7647 px
Network packets	123850
Size in network	34.74 MB
Average image network size	4.10 KB
Network overhead	46.0%
Average network bitrate	0.46 Mbps
Estimated IP camera bitrate	4 Mbps
Estimated IP camera video size	301 MB
Bandwidth reduction factor (factor that divides)	8.7

Table 9: Test Results – Example of High Density

The bandwidth reduction factor in Table 9 obtained in the high density test shows an important bandwidth reduction, although not of the same magnitude as the medium density test.

6.9.3 Very High Density

The video from [59] has a facial density that falls into the very high density category, given that after extracting the faces from the video, as shown in Figure 65, the average number of face images detected and extracted per minute has been determined to be 1479.2. The data used to obtain this metric can be found in Table 10. Following the same procedure as in the medium density test, the results of the test with the very high density video are shown in Table 10**.**

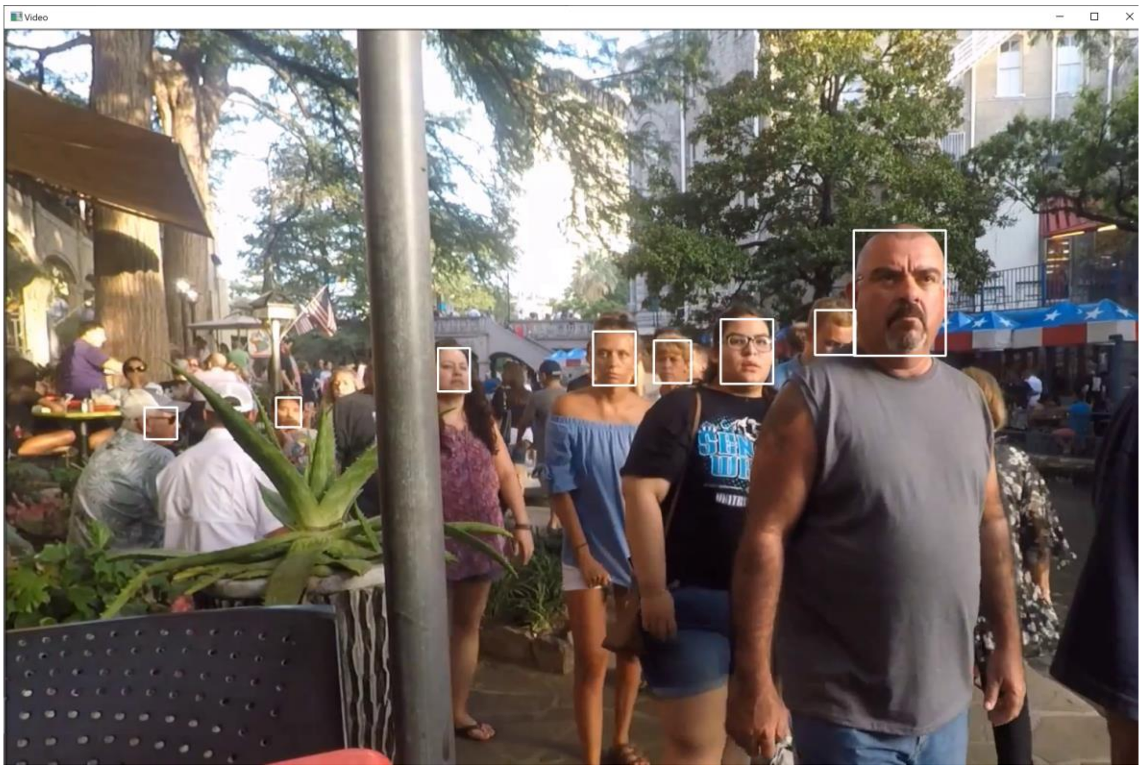


Figure 65: Face Extraction Window – Example of Very High Density

Video length	601 s
Number of face images	14817
Face images per minute	1479.2
Total face image files size	51.01 MB
Average image file size (kb)	3.53 KB
Average image height	91 px
Average image width	74 px
Average image size (pixels)	7755 px
Network packets	221712
Size in network	70.22 MB
Average image network size	4.85 KB
Network overhead	37.7%
Average network bitrate	0.93 Mbps
Estimated IP camera bitrate	4 Mbps
Estimated IP camera video size	300.5 MB
Bandwidth reduction factor (factor that divides)	4.3

Table 10: Test Results – Example of Very High Density

The bandwidth reduction factor in Table 10 obtained in the very high density test shows a remarkable bandwidth reduction, far off the result of the medium density test as expected, but still providing more than a four times reduction in the bandwidth required, even in such a challenging environment.

6.9.4 Discussion of Results

The results of the bandwidth reduction tests conducted in the previous subsections have led to the obtention of the bandwidth reduction factors shown in Table 11. The bandwidth reduction factor is represented as dependent on the number of faces detected and extracted by the edge program per minute. The metric also depends on the average size of the faces extracted.

Range	**Face images per minute**	**Expected bandwidth reduction factor**
Very low density	<5	>1365x
Low density	5-50	1365x-137x
Medium density	50-500	137x-14x
High density	500-1000	14x-7x
Very high density	>1000	<7x

Table 11: Face Density Ranges and Expected Bandwidth Reduction

The bandwidth reduction estimation shown in Table 11 has been obtained considering a 4 Mbps camera stream and an average 4.5 KB per face image transmitted. The average image size has been obtained as an average of the tests shown in this section and of other tests conducted during the development of this project.

Chapter 7: BRIEF CYBERSECURITY ANALYSIS

The architecture designed in this project is mainly focused on the reduction of bandwidth through the use of edge computing. Even so, attention has also been paid to system cybersecurity, as demonstrated by some of the attack mitigations incorporated in the code. Nevertheless, not being the main target, there are some security protections that have not been considered, as they are either implementation dependent or application dependent. Due to this, while several security functions and vulnerability and exploitability mitigations have been implemented, there are some basic features that should be additionally incorporated prior to the actual deployment of the system in a production environment, especially when the data transmitted to and from the Cloud API have to be confidential.

7.1 EXPLOITABILITY MITIGATIONS

In some of the Calls of the Cloud API, the inputs provided in the request fields are used to access the filesystem. This means that if left untreated, inserting expressions such as "../" into those fields could allow attackers to access areas of the filesystem to which they are not authorized, as they are out-of-bounds. This has been solved by checking that the path to which the functions that handle user-provided inputs access is not out of the scope of their access (e.g. the Dataset folder in the case of adding or deleting IDs).

In the Cloud API, some of the requests handle user inputs and use them to send an SQL query to the Database server. These inputs cannot be directly inserted into an SQL query, as this would open a door for SQL injections attacks. This type of attacks perform unauthorized access or modification of the database by inserting special commands into the user inputs that are later included in the SQL query. This enables attackers to read or make changes outside of the scope of the query.

To block the SQL injection attacks, the system parses the inputs via the mysql connector's built-in functions. These functions take the query template and the user-defined input parameters in different variables, so that they can check that there are no special characters or illegal expressions in the user-provided inputs.

In the cloud infrastructure, the Virtual Machine (VM) that runs the Network File Storage (NFS) service is accessible through SSH for development purposes. In a production environment, this connection should be made through a Virtual Private Network (VPN) instead of over the internet, where anyone can try passwords. However, to partially mitigate this, instead of authenticating with a user-password combination, the access authentication is performed with a certificate, increasing the security of the system.

7.2 FURTHER CYBERSECURITY IMPROVEMENTS

As previously mentioned, there are some cybersecurity mitigations and cybersecurity vulnerability patches that have not been implemented in this project since they are implementation dependent. This means that they would have had to be implemented again in any new deployment.

The Cloud API is designed as a RESTful API. Due to this, the requests are sent with the HTTP protocol. This protocol is not secure and is vulnerable to both confidentiality and integrity losses. The solution to this is to use HTTP over Transport Layer Security (TLS). This combination is known as the HTTP Secure (HTTPS) protocol. To implement it, the Cloud API servers should be configured with a security certificate for the connection, and HTTPS capabilities should be enabled in the Web Server handling the API Calls.

The connection between the Edge Nodes and the Cloud API, as well as the connection between the programs used by the employees performing maintenance operations (e.g. user add and user delete operations) and the Cloud API is not authenticated. This lack of authentication means that anyone with access to the Public IP address used as the ingress for the API is able to interact with it. This may not be optimal for a production deployment, and device and employee authentication should be implemented.

Both the HTTPS security and the device/employee authentication could be implemented by adding an extra authentication and unpacking deployment before the currently implemented processing deployment. This deployment would handle both the authentication and the HTTPS requests. It would then send the extracted HTTP requests to the processing deployment. This way, the processing deployment would perform the same operations as in the current system. This would also allow to add an extra division of the processing pods, between face recognition handlers and training and dataset maintenance handlers, as the intermediate deployment could determine to which it should send the HTTP request depending on its URL.

The API calls to the Cloud API require a significant processing time (given the complexity of the service being provided) to generate an answer when compared with an average HTTP or PHP webpage server. This means that a Distributed Denial of Service (DDoS) attack would not require much traffic to achieve its goal of taking down the service (or generating a great traffic spike which would lead to high Cloud Computing billing if using auto-scaling services, an option of the proposed architecture). This would be greatly mitigated with the aforementioned user authentication, as the DDoS could be contained into the authentication phase which is orders of magnitude faster than the processing phase.

Finally, both the libraries used in the project and the code that has been developed should be periodically checked for new security updates and any vulnerabilities that may be discovered by the international community and may affect the code.

Chapter 8: BRIEF LEGAL ANALYSIS

The popularization of the use of facial recognition systems is generating controversy all around the world. Legislation is being implemented to protect face biometrics as much as other types of biometrics such as DNA and fingerprints are already being protected.

Faces are public identifiers that humans use daily to recognize the people in their environment. Face recognition systems use the same information as humans do to identify people in pictures and video feeds. Therefore, faces can be considered personal identifiers and, as such, should be protected by the privacy legislation of countries.

8.1 EUROPEAN UNION'S GENERAL DATA PROTECTION REGULATION

According to the General Data Protection Regulation (GDPR), active since 2018 in the European Union, face images are considered biometric data. Biometric data are considered sensitive personal information by the GDPR. As sensitive information, face images are subject to a high level of legal protection, which requires the highest level of data protection.

Furthermore, under the protection provided by the sensitive personal data category of the GDPR, the use of the facial features of a person is highly restricted. It is protected to the extent that, in most cases, these characteristics can only be used after user consent has been provided to the processor of the information. This fact leads to the use of facial recognition software almost exclusively by governmental agencies, where its use can be justified as of importance for public safety. For this reason, facial recognition is commonly found in law enforcement use.

8.2 UNITED STATES

In the United States, facial recognition software is being used by many law enforcement agencies, both federal and state ones. This massive use of facial recognition is backed by an extensive network of databases that contain pictures of hundreds of millions of people. The storage of this information over many departmental databases poses a severe risk to individual privacy due to a high risk of information leakage and potential misuse.

Many senators, legislators and public leading figures have spoken out against this indiscriminate use of facial recognition software in the US. They allege that it not only attacks individual privacy, but also creates new problems; problems such as false positives that may lead to the arrest of innocent individuals or the bias of certain systems when dealing with people of different race and skin color.

Although there is no major federal legislation regarding the use of facial features for face recognition, there are state laws that regulate the use of biometric identifiers, such as the Illinois Biometric Information Privacy Act (BIPA) [60]. This law protects biometric identifiers from being obtained or stored by companies without the previous explicit consent of the citizen.

In the Illinois BIPA, biometric identifiers are defined as "retina or iris scan, fingerprint, voiceprint, or scan of hand or face geometry". It is noteworthy that biometric identifiers do not include "writing samples, written signatures, photographs, human biological samples used for valid scientific testing or screening, demographic data, tattoo descriptions, or physical descriptions such as height, weight, hair color, or eye color".

Face recognition systems would fall under the definition of “face geometry” biometric identifiers and, as such, the companies that intend to use them would require the explicit consent of the user/citizen being scanned, under the Illinois BIPA.

Chapter 9: CONCLUSIONS

After the completion of this project, the following conclusions can be drawn regarding the system design:

1. A face recognition architecture optimized for low bandwidth between the installation site and the cloud, low processing power in the edge, high availability and high scalability in the cloud has been designed.
2. A Cloud API has been designed in Python to accommodate the face recognition procedures, following RESTful naming guidelines and best practices.
3. The deployment has been realized so that it can be rapidly scaled up to running hundreds of instances of the Cloud API code if necessary, which allows for both high availability and scalability.
4. The system has been designed so that new faces can be added to the list of people that can be recognized without any face recognition downtime while training the system for the new people.
5. The training time of the classifier has been significantly reduced by storing and maintaining updated the embeddings extracted from the known images of the individuals that can be recognized by the system.
6. The system has been prepared to be able to work both standalone and in cooperation with a mobile APP that provides valuable statistics to the user.
7. A brief cybersecurity analysis with security considerations and maintenance recommendations has been provided for future implementations.
8. A brief legal analysis of face recognition systems has been carried out to give a fundamental view regarding deployment considerations in both the US and EU.

Furthermore, the following conclusions can be drawn with respect to the system implementation:

9. The Cloud API has been implemented in a Kubernetes deployment in the Microsoft Azure cloud, which is accessible through the internet.
10. The system has been equipped with shared network storage to allow for scalability.
11. A database has been integrated as one of the destinations of the detections done by the system to allow for further analysis.
12. A series of programs have been developed to serve as edge computing to detect faces from a variety of sources (web camera, video file, video stream) and send them to the Cloud API.
13. All programs that interact with the Cloud API to recognize the faces in a web camera, camera signal, video stream or video file include options to configure which frames to process, which face images to send, and how much to compress them before being sent.
14. The recognition requests for the faces detected in a given frame can be sent simultaneously for reduced frame processing times, in all the programs that interact with the Cloud API to recognize faces.
15. Several programs have been integrated to allow for the system management from the employee’s point of view, with routines that enable adding new people to the recognition system and adding new reference images from different sources.
16. An ID can be completely removed from the system with only two API calls leaving no trace in the storage or the recognition system for improved legal compliance.

The results of the testing conducted lead to the following conclusions:

17. The correct functionality of the system has been demonstrated with a series of tests designed to cover the most relevant functionality.
18. The performance of the system has been measured, and its scalability has been tested.
19. The bandwidth reduction, which was the primary motivation and objective of the system, has been proven in a test scenario.

As a summary, we can affirm that the system implemented has proved that the architecture proposed in this project achieves what it was originally designed for: reducing the bandwidth in face recognition of video surveillance systems, thanks to the use of a design based on Edge Computing. Furthermore, it achieves scalability and high availability thanks to the implementation on a Kubernetes cluster.

Chapter 10: Future Work

The following work is proposed for future extensions of this project, that can improve the application to real cases, with advanced performance and user interaction:

1. Train a new face feature extractor based on the newest and most efficient architectures such as MobileNet V3 or EfficientNet. This would allow the system to recognize the faces with improved accuracy and efficiency.
2. Further optimize the classifier for large number of IDs, as the current version works great for small to medium sizes but may struggle with hundreds or thousands of IDs.
3. Optimize the face detection routine of the Cloud API to obtain even higher performance per CPU core, although the performance is already good for the complexity of the operations being performed.
4. Further reduce the bandwidth of the edge computing nodes to the Cloud API by removing remarkably similar face images of the same person from frame to frame. This can be achieved by tracking the faces and not sending similar images of the same face.
5. Implement the edge processing in a specific device as the ones suggested in Chapter 2. This would allow to speed-up the edge processing by taking advantage of the hardware acceleration provided by GPUs and TPUs.
6. Implement an even faster face detection algorithm in the edge, as this algorithm is to be optimized towards speed and not accuracy, taking into account that false positives are not a problem, given that the accuracy is provided by the second face detection, which is performed in the cloud.
7. Divide the work being performed by the face recognition Pod between face recognition and dataset and model management.
8. Implement edge node and employee program authentication.
9. Implement HTTPS security with a security certificate from a well-known Certification Authority (CA).
10. Implement Distributed Denial of Service protection or mitigation in the Cloud API for improved robustness.
11. Detect the approximate distance between people in the camera frames using face size and distance between faces for certain applications, like COVID infection prevention.
12. Design an analysis program that uses the detections information in the database to create processed high-level information.
13. Design an algorithm that groups unknown faces based on embedding similarity allowing to add new people from the camera feeds and enabling to track unknown people with auto-generated identities.

REFERENCES

[1] Cisco Systems, "Cisco Annual Internet Report (2018–2023) White Paper," 2018.

[2] "Edge Computing Trends for 2020 & the Next Decade," 2020. [Online]. Available: https://datacenterfrontier.com/edge-computing-trends-2020/. [Accessed 09 05 2020].

[3] Ring Inc., “Ring Home Security,” [Online]. Available: https://www.ring.com/. [Accessed 23 07 2020].

[4] Google LLC, “Nest Connected Home,” [Online]. Available: https://nest.com/. [Accessed 23 07 2020].

[5] Xiaomi, “Xiaomi Global,” [Online]. Available: https://www.mi.com/global. [Accessed 23 07 2020].

[6] B. Amos, “OpenFace,” [Online]. Available: https://cmusatyalab.github.io/openface/. [Accessed 06 08 2020].

[7] F. Schroff, D. Kalenichenko and J. Philbin, "FaceNet: A Unified Embedding for Face Recognition and Clustering," arXiv, 2015.

[8] D. Sandberg, “A FaceNet Implementation,” [Online]. Available: https://github.com/davidsandberg/facenet. [Accessed 06 08 2020].

[9] Tensorflow, “Tensorflow: an end-to-end open source machine learning platform,” [Online]. Available: https://www.tensorflow.org/. [Accessed 06 08 2020].

[10] Arsfutura, “A framework for creating and using a Face Recognition system.,” [Online]. Available: https://github.com/arsfutura/face-recognition/tree/669b625bc5f12e79baa4bc662060279aa4d2ab6c. [Accessed 06 08 2020].

[11] Timesler, “Pretrained Pytorch face detection (MTCNN) and recognition (InceptionResnet) models,” [Online]. Available: https://github.com/timesler/facenet-pytorch. [Accessed 06 08 2020].

[12] D. Sandberg, “Face Detector Python Code,” [Online]. Available: https://github.com/davidsandberg/facenet/blob/master/src/align/detect_face.py. [Accessed 23 07 2020].

[13] D. Sandberg, “Inception ResNet V1 Python Code,” [Online]. Available: https://github.com/davidsandberg/facenet/blob/master/src/models/inception_resnet_v1.py. [Accessed 23 07 2020].

[14] C. Cortes and V. Vapnik, Support-Vector Networks, Boston: Kluwer Academic Publishers, 1995.

[15] C. Mateo Muñoz, “Design and development of mobile applications for the analysis and visualization of data about the position of people in a location,” Chicago, 2020.

[16] H. Anton, Elementary Algebra, John Wiley & Sons, 1994.

[17] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician,* vol. 3, no. 46, pp. 175-185, 1992.

[18] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen and M. Tan, "Searching for MobileNetV3," arXiv, 2019.

[19] ImageNet organization, "ImageNet," [Online]. Available: http://www.image-net.org/. [Accessed 10 05 2020].

[20] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *International Conference on Machine Learning*, Long Beach, California, 2019.

[21] C. Szegedy, S. Ioffe, V. Vanhoucke and A. A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," in *Procedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[22] NVIDIA Corporation, "NVIDIA," [Online]. Available: https://www.nvidia.com/. [Accessed 10 05 2020].

[23] NVIDIA Corporation, "NVIDIA Jetson," [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/jetson-store/. [Accessed 10 05 2020].

[24] International Telecommunication Union, High Efficiency Video Coding H.265, ITU-T, 2019.

[25] Amazon Incorporated, "Amazon Web Services," [Online]. Available: https://aws.amazon.com. [Accessed 10 05 2020].

[26] Alphabet Incorporated, "Google Cloud Computing Services," [Online]. Available: https://cloud.google.com. [Accessed 10 05 2020].

[27] Microsoft Corporation, "Azure Cloud Services," [Online]. Available: https://azure.microsoft.com/en-us/. [Accessed 05 08 2020].

[28] VMware Incorporated, "ESXi Hypervisor," [Online]. Available: https://www.vmware.com/products/esxi-and-esx.html. [Accessed 05 08 2020].

[29] Docker Incorporated, "Docker Hub," [Online]. Available: https://hub.docker.com/. [Accessed 05 08 2020].

[30] Microsoft Corporation, "GitHub," [Online]. Available: https://github.com/. [Accessed 05 08 2020].

[31] Ars Futura, "Face Recognition Framework," [Online]. Available: https://github.com/arsfutura/face-recognition. [Accessed 24 07 2020].

[32] Fyr91, "Face Detection Python Program," [Online]. Available: https://gist.github.com/fyr91/79aaf4b6d679814406ee4028bd03b7aa. [Accessed 24 07 2020].

[33] NumPy, “Numpy: The fundamental package for scientific computing with Python,” [Online]. Available: https://numpy.org/. [Accessed 24 07 2020].

[34] F. Lundh and A. Clark, “The Pillow Library,” [Online]. Available: https://python-pillow.org/. [Accessed 24 07 2020].

[35] Timesler, “Facenet Pytorch library,” [Online]. Available: https://github.com/timesler/facenet-pytorch. [Accessed 24 07 2020].

[36] Pytorch, “Pytorch: Tensors and Dynamic neural networks in Python,” [Online]. Available: https://github.com/pytorch/pytorch. [Accessed 24 07 2020].

[37] Pytorch, “TorchVision: Datasets, Transforms and Models specific to Computer Vision,” [Online]. Available: https://github.com/pytorch/vision. [Accessed 24 07 2020].

[38] OpenCV team, “OpenCV: Open source computer vision and machine learning software library,” [Online]. Available: https://opencv.org/. [Accessed 24 07 2020].

[39] The Scikit Community, “Scikit Learn Machine Learning in Python,” [Online]. Available: https://scikit-learn.org/stable/index.html. [Accessed 24 07 2020].

[40] Joblib, “Joblib library,” [Online]. Available: https://github.com/joblib/joblib. [Accessed 24 07 2020].

[41] Pallets, “Flask library,” [Online]. Available: https://github.com/pallets/flask/. [Accessed 24 07 2020].

[42] Noirbizarre, “Flask RESTPlus library,” [Online]. Available: https://github.com/noirbizarre/flask-restplus. [Accessed 24 07 2020].

[43] SMARTBEAR, “Swagger API development and documentation tool,” [Online]. Available: https://swagger.io/. [Accessed 24 07 2020].

[44] Unbit, “uWSGI Application Server Container,” [Online]. Available: https://github.com/unbit/uwsgi. [Accessed 24 07 2020].

[45] Pallets, “Werkzeug: The comprehensive WSGI web application library,” [Online]. Available: https://github.com/pallets/werkzeug/. [Accessed 24 07 2020].

[46] PSF, “Requests library: A simple yet elegant HTTP library,” [Online]. Available: https://github.com/psf/requests. [Accessed 24 07 2020].

[47] Oracle Corporation, “MySQL Connector/Python Developer Guide,” [Online]. Available: https://dev.mysql.com/doc/connector-python/en/. [Accessed 09 08 2020].

[48] The Matplotlib development team, “Matplotlib library: Visualization with Python,” [Online]. Available: https://matplotlib.org/. [Accessed 24 07 2020].

[49] The Linux Foundation, “ONNX: The open standard for machine learning interoperability,” [Online]. Available: http://onnx.ai/. [Accessed 24 07 2020].

[50] spyoungtech, “Grequests: Requests + Gevent library,” [Online]. Available: https://github.com/spyoungtech/grequests. [Accessed 01 08 2020].

[51] Gevent, “Gevent: Coroutine-based concurrency library for Python,” [Online]. Available: https://github.com/gevent/gevent. [Accessed 01 08 2020].

[52] UMASS, “Labeled Faces in the Wild Dataset,” [Online]. Available: http://vis-www.cs.umass.edu/lfw/. [Accessed 01 08 2020].

[53] S. Spielberg, Director, *Indiana Jones and the Kingdom of the Crystal Skull.* [Film]. United States: Lucasfilm Ltd., 2008.

[54] IPVM, “Report on bandwidth for IP video surveillance systems,” [Online]. Available: https://ipvm.com/reports/bandwidth-tutorial-for-ip-video-surveillance-systems. [Accessed 09 08 2020].

[55] Reolink, “IP camera bandwidth calculation,” [Online]. Available: https://reolink.com/ip-camera-bandwidth-calculation/. [Accessed 9 08 2020].

[56] Watched Walker, “London Walk Test Video,” [Online]. Available: https://youtu.be/ETZ3kHa6xc0?t=30. [Accessed 15 09 2020].

[57] Wireshark Foundation, “Wireshark Network Analyzer,” [Online]. Available: https://www.wireshark.org/. [Accessed 15 09 2020].

[58] Exploring Alabama, “Pier park Panama city test video,” [Online]. Available: https://youtu.be/nFqtN2ri6fA?t=1024. [Accessed 15 09 2020].

[59] Exploring Alabama, “San Antonio river test video,” [Online]. Available: https://youtu.be/c0GlCjZbVzk?t=901. [Accessed 15 09 2020].

[60] The State of Illinois, “Illinois Biometric Information Privacy Act,” [Online]. Available: https://www.ilga.gov/legislation/ilcs/ilcs3.asp?ActID=3004&ChapterID=57. [Accessed 01 08 2020].