Bruno de Sousa Bastos

Serviços de Computação para Análise de Sentimentos em Sistemas de Vídeo Conferência

Computing Services for Sentiment Analysis in Videoconferencing Applications

Universidade de Aveiro
2023

Bruno de Sousa Bastos

Serviços de Computação para Análise de Sentimentos em Sistemas de Vídeo Conferência

Computing Services for Sentiment Analysis in Videoconferencing Applications

Proposta de Tese apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à conclusão da unidade curricular Proposta de Tese, condição necessária para obtenção do grau de Mestre em Engenharia Informática , realizada sob a orientação científica do Doutor Ilídio Castro Oliveira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e da Doutora Susana Manuela Martinho dos Santos Baía Brás, Professora investigadora no Instituto de Engenharia Eletrónica e Informática da Univerisadade de Aveiro.

o júri / the jury

presidente / president

Prof. Sérgio Guilherme Aleixo de Matos
Professor Auxiliar em Regime Laboral, Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Rolando da Silva Martins
Professor Auxiliar, Universidade do Porto - Faculdade de Ciências

Prof. Doutor Ilídio Fernando de Castro Oliveira
professor associado da Universidade de Aveiro

Palavras Chave Vídeo Conferência, Reconhecimento de Emoções, MLOps, DevOps, Containers

Resumo

Durante a pandemia muitas organizações optaram por um regime de teletrabalho de maneira a garantir a continuidade do negócio e o bem estar dos trabalhadores. A flexibilidade proporcionada pelas tecnologias de colaboração remota provaram ser vantajosas, tornando-se um meio de comunicação bastante utilizado no pós-pandemia. Visto isto, notou-se que o reconhecimento das emoções dos seus participantes é um tópico com caminhos inexplorados. Recentemente houve um enorme progresso na capacidade de classificar corretamente emoções humanas por parte dos algoritmos de machine learning. Ao aplicarmos esses algoritmos a uma platforma de video conferência seria possível analisar em tempo real as emoções dos seus participantes. O objetivo principal deste trabalho é a criação de um sistema de prova de conceito capaz de integrar modelos de machine learning numa plataforma de vídeo conferência, prorcionando a análise das emoções através de áudio e vídeo. O sistema passou por uma fase de conceção que visava ultrapassar problemas relacionados com a sua escabilidade e flexibilidade. A arquitetura resultante é composta por vários componentes que permitem a criação de uma platforma de vídeo conferência, a captura, agregação e posterior análise dos dados de áudio e vídeo dos participantes. Foram criados dois deployments, local em Docker, e um em Kubernetes. Aspetos como a monitorização do sistema foram também abordados. No sistema prova de conceito é possível obter a classificação das emoções através do vídeo e do aúdio em quase tempo real. Os testes e posterior resultados mostraram que dependendo dos modelos usados e das suas configuções é possível minimizar o tempo de retorno das classificações aos participantes.

Keywords Video Conference Platform, Emotion Recognition, MLOps, DevOps, Containers

Abstract

During the pandemic many organizations adopted the remote work as way of ensuring the work continuity and the well being of its workers. The flexibilty provided by remote collaboration tools proved to be advantageous, becoming a prominent way of communication between teams. Nevertheless, the ability to understand the participants' emotions is a topic yet to fully explore. Recent advancements in machine learning algorithms have reached newer milestones in their ability to accurately classify human emotions. By applying this algorithms to a video platform it would be possible to analyse in real time the emotions of its participants. The main objective of this work is the creation of a proof of concept system, capable of integrating machine learning models in a video conference platform, providing analysis of emotions based on audio and video. The system has gone through a conceptual phase to overcome challenges related to its scalability and flexibility. The resulting architecture is made of multiple components that allow the creation of video conferences, the capture, aggregation and later analysis of the participants audio and video data. Two deployments were created: Docker and Kubernetes. The monitorization of the system was also explored. In this proof of concept system it is possible to obtain the classified emotions based on the participant's audio and video in near real-time. The tests and posterior results showed that depending on the models used and on their configurations it is possible to minimize the delay for the classifications to reach the participants.

Contents

List of Figures

List of Tables

List of Code Snippets

Glossary

WebRTC	Web Real-Time Communication
SDP	Session Description Protocol
API	Application Programming Interface
ICE	Interactive Connectivity Establishment
SFU	Selective Forwarding Unit
MCU	Multipoint Conferencing Unit
P2P	Peer to Peer
AWS	Amazon Web Services
MLOps	Machine Learning Operations
NAT	Network Address Translation
STUN	Session Traversal Utilities for NAT
TURN	Traversal Using Relays around NAT
CNN	Convolutional Neural Network
KMS	Kurento Media Server
XGBoost	eXtreme Gradient Boosting

CHAPTER 1

Introduction

1.1 Motivation

The adoption of video conference systems has increased since its peak usage during the Covid-19 pandemic. Remote work was widely adopted by many organizations to ensure business continuity and their employees' well-being during the quarantine period. The flexibility of remote collaboration proved advantageous even after the health crisis, becoming a prominent way of communication between teams. Nevertheless, the ability to understand the participants' emotions is a topic that has yet to be fully mapped.

Emotion recognition and sentiment analysis aim to understand a person's behaviours and decision-making. Recent advancements in machine learning algorithms have reached newer milestones in their ability to accurately classify human emotions[1]–[3]. Those models can operate on audio and video data, which can be found aplenty on video conference systems. By harvesting the data from video conferences and provisioning it to the machine learning models, it is possible to predict the participants' emotions in real-time.

Machine learning models for emotion recognition in a video conference environment have two main scenarios. Offline analysis is when the models are used for classification after the meeting has ended. In online or real-time analysis, the predictions are made in an ongoing meeting, providing near real-time feedback to the participants. This work focuses on the latter by exploring the possibility of adding near real-time analysis to a video conference environment.

Due to the real-time nature of video conferences, the prediction process needs to be fast and scalable to accommodate a possible upsurge in the number of participants using the system. With such requirements, the system architecture needs to be well conceived and designed towards performance and flexibility in adapting newer models.

In summary, the research focuses on applying the discoveries on emotion recognition to a real-time environment based on video conferences. The detection of the emotional state of the participants could help the host dynamically adopt different strategies that seem more fitting.

1.2 Objectives

The main objective of this work is to develop a proof of concept extension to video conference systems to include near real-time feedback on sentiment analysis of participants, using audio and video. The system has to be capable of hosting a video conference and providing real-time feedback for its participants about the detected emotions. The development of the methods for emotion classification is out of the scope of this project. To our objectives, it will be sufficient to select and incorporate available machine learning models.

An architecture has to be conceived to connect the video conference component to the machine learning classifiers, providing audio and video data in near real-time. An effort has to be made to guaranteed that the addition of other classifiers is possible and simplified. Assuring the monitorization of the classifiers and the other components of the system is another aspect that concerns this work.

In the early stages, the system should have the capability of operating locally, and further into the development should be integrated into an environment where it can be scaled. This will provide ways of testing the system's ability to scale and adapt to the load.

1.3 Challenges

The creation and implementation of a scalable video conferencing system is a significant challenge. Users from different parts of the globe expect seamless communication with high video and audio quality. Additionally, the emotion recognition model must process large volumes of continuous user-generated data in real time. Any delay in this process can disrupt user immersion and lead to outdated or inaccurate results.

Those system characteristics create many challenges based on performance and delay. Implementation-wise, video conference systems can be hard to execute properly, relying on recent and complex technologies. On top of that, the usage of models that might not be suitable for the real-time environment might prove to be a challenge down the line. The work might also suffer from hardware constraints by being limited to a single testing and development machine.

The usage of different classifiers can also create some problems, as each one of them might require distinct inputs. That can lead to different ways of partitioning the video and audio according to the model specifications. Some can make predictions on a single frame, while others are based on time windows.

Video platform participants' audio and video data is sensitive. In some scenarios, there might arise some problems due to the privacy of the data and privacy protection mechanisms may be required.

CHAPTER 2

Background and State of the art

An overview of the state-of-the-art video conferencing systems is provided, focusing on their deployment architecture and the employed scalability and fault tolerance techniques.

Afterward, we explored the state-of-the-art machine learning models for emotion recognition for audio and video features. This focuses on understanding the models at a high level, giving special attention to how they interact with the input features.

Subsequently, the machine learning model serving is explained, enumerating the multiple components that characterize MLOps. It includes a general description of the methodologies and technologies used, leaning towards emotion recognition in video conference systems.

Thereafter, the needs and techniques for system deployment and scalability are depicted in conjunction with the approaches for system monitoring.

2.1 Video Conference Systems

The internet brought the ability for users to communicate while being far apart. During the Covid-19 pandemic, this possibility became necessary for the world to keep moving. Video conference platforms rose in popularity, becoming an alternative for face-to-face communications and maintaining some relevance even after the restrictions' removal[4]. Built upon a well-thought-out architecture, these systems are complex and rely on many components to provide the users with a good experience that can have advantages over face-to-face communications.

2.1.1 Web Real-Time Communication

WebRTC[5]–[7] was created by Google in 2011 to facilitate P2P communications over the web. Since then, it has evolved and become a well-established technology for video conference systems[8], [9]. The technology provides an Application Programming Interface (API) available in Javascript and as a native library for mobile platforms such as Android and IOS. It is open source, is constantly evolving, and is present in most modern browsers. Overall WebRTC

facilitates the development of real-time media communications due to its high level API and the majority of the clients' devices have it built-in[10].

The API provides the essentials for developers to start communications between peers. In a WebRTC connection, peers exchange data through media streams. Streams represent a link between two peers that can have multiple tracks, each transmitting a data type. Audio and video are the most common data types supported.

P2P communication allows peers to communicate with each other without needing a third member to route the traffic between them. The connection is guaranteed to be private and secure.

However, peers require each other's information to establish the link. This process relies on an external server called a signaling server. This server is known by both peers and works as a proxy. WebRTC protocol does not specify the communication protocol with the signaling server. Websockets and HTTP requests are the most common choices used by developers.

The communication setup has two crucial steps that can be done in parallel due to the asynchronous behavior of modern browsers. Initially, one of the peers sends an offer containing the media types and the respective codecs it is willing to receive or send. This information is designated Session Description Protocol (SDP). Upon receiving this offer, the other peer generates an answer with similar information. The browsers implement and use the codecs to compress audio and video segments, increasing the transmission speed by reducing the amount of information sent. If, for any reason, one of the peers does not support any of the codecs in the offer, the peers will not be able to establish the connection. After a successful negotiation, both peers can understand each other.

While the offer and answer are still in negotiation, peers, in parallel, can start gathering Interactive Connectivity Establishment (ICE) candidates. They offer guidance on how to reach the local peer, detailing the path through devices to reach it. The other peer receives the candidates as they gather.

Once both peers have acquired the ICE candidates from each other and exchanged the SDP offer and answer, they can establish communication without any external intervention. After this point, the signaling server is no longer necessary, and the communication between the peers is private and secure.

STUN server

WebRTC connections can be established smoothly if both participants are in the same network. In a real scenario, peers report their private network addresses. Network Address Translation (NAT) provides those addresses which are unusable outside the local network. When communicating with external networks, the NAT will translate private addresses to public addresses and vice versa. STUN servers live on the internet and have the task of responding to a request at the address and port of the requester. Peers send that information over the signaling server and can connect afterward.

TURN server

Restrictions in the form of firewalls and internal company networks can prevent access to specific IP addresses. When unique identifiers are not present, it becomes difficult for networks to connect. This situation is overcome by employing a middleman known as a TURN server. Unlike STUN, a TURN server stays active in the communication path even after establishing a connection. However, this solution can cause delays in video conferencing and requires a server capable of relaying the streams from all the connected peers.

Figure 2.1 shows an example of how to set up a WebRTC connection and its intervenients.

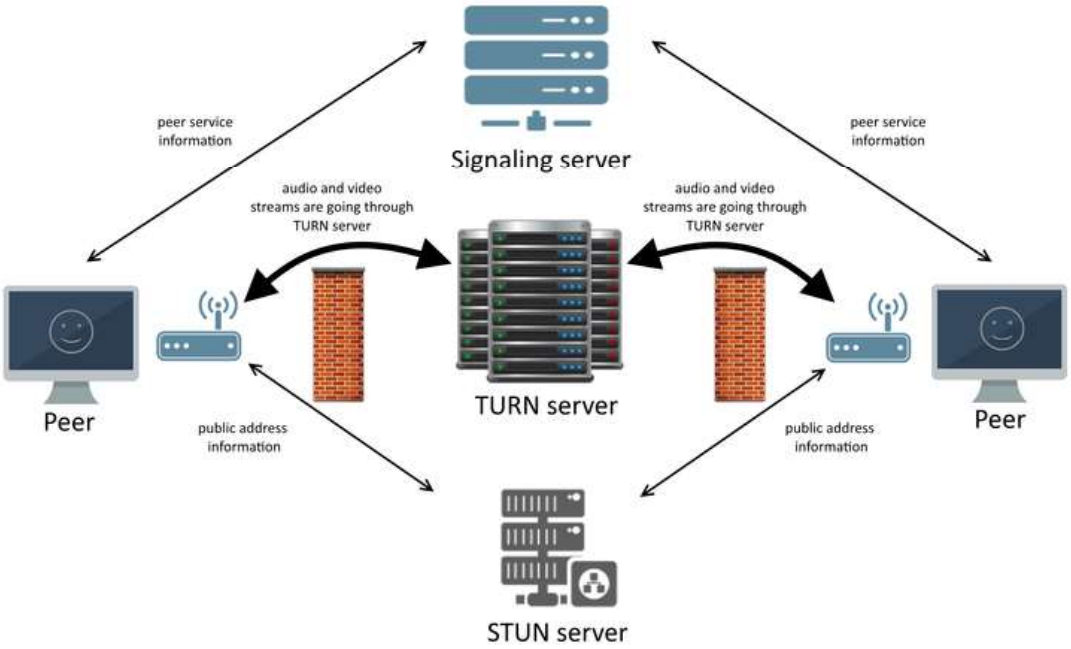

Figure 2.1: Example of two peers establishing a WebRTC connection. The public address is obtained using a STUN server. Data is exchanged using the signaling server. After the WebRTC connection is established the TURN server relays the traffic.[11]

2.1.2 Media Server

In video conferences where many participants are communicating with each other, the use of a P2P connection compromises the performance and video quality of the calls. Due to that, developers started using a proxy server that forwards the media between peers. There are multiple media server implementations and some topologies appeared based on them. The most notorious topologies are Multipoint Conferencing Unit, and SFU[12]–[14]. Some approaches consist of hybrid topologies.

In P2P Mesh topologies, each participant creates a connection with everyone on the call except themselves. This topology does not make use of a media server and reflects the basic scenario of P2P connection. Data is encoded and sent to all the participants putting a strain on the uplink network. Uplink bandwidth rapidly becomes insufficient as more participants connect. It becomes a consistent bottleneck in this topology that makes it unable to scale.

Due to how WebRTC works, every link can have a different codec. When sending data to other participants, the sender's CPU has to encode the video and audio stream multiple times, overloading the CPU. Receiving data is less of a problem for the bandwidth because most networks have access to more downlink bandwidth than uplink. The CPU problem persists but to a lesser degree since decoding is faster and lighter than encoding. For those reasons, P2P Mesh topology is unreliable and is not recommended for group calls of more than three participants. Nevertheless, it might see usage in some scenarios because the connections are fast and private since there is no intermediary.

Figure 2.2 provides an overview of the topology of the connections when there are five participants.

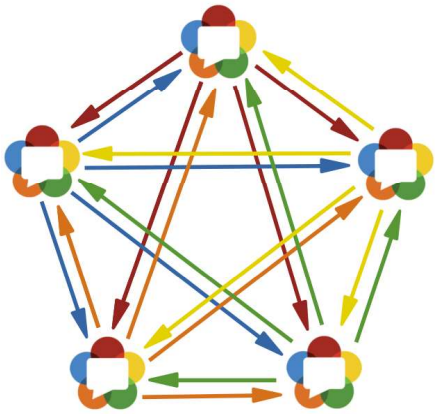

Figure 2.2: Example of WebRTC P2P Mesh topology with five participants.[13]

In MCU topology, every peer only has two connections, one to send and another to receive data. Every participant connects to a media server called MCU. Upon receiving data from every stream, the server mixes them and sends the mixed data to every participant. A small number of connections decreases the processing made on the client side, and uploading bandwidth is not a problem anymore. However, this solution requires a powerful server to ensure the mixing of the streams with minimal latency. The hardware limits the number of participants, making this solution unable to scale to host significant conferences. Mixing the streams into a single one reduces each participant's control over the audio and video individual streams of a specific participant. It disables muting, increasing/decreasing volume, turning off the camera, and many others on a participant level. Figure 2.3 pictures an example of a group call with five participants using the MCU topology.

SFU works as a media server, receiving and forwarding the media from one participant to another, but does no mixing to the incoming media. The SFU connects the incoming media stream to the many output streams. If a participant wants to send data, it will do so through a single connection to the media server. To receive data, it needs a stream for every other participant. This media server allows for more customization since it can configure the stream individually. Since the streams are not mixed either, the latency is reduced, detrimental to a more intensive CPU task on the client side. Figure 2.4 presents an example of the topology.

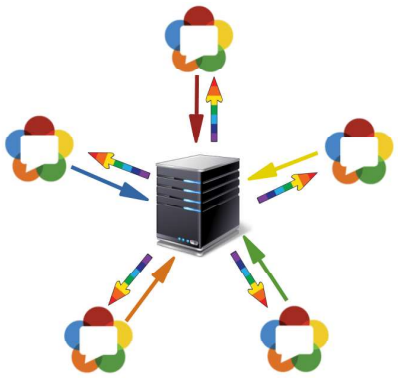

Figure 2.3: Example of a WebRTCMCU topology with five participants.[13]

Figure 2.4: Example of a WebRTC SFU topology with five participants.[13]

This freedom provided by the SFU media server type and its performance benefits over the other types is why it is the preferred type for video conference platforms. Free media server implementations are available for use that hide the complex logic behind a more easy-to-use API.

In the table 2.1 there is a general comparison of the different topologies. The symbols in the table represent, in comparison, if the value is higher or lower than the others. The colors identify whether that value is good or bad.

Table 2.1: WebRTC topologies comparison.

	P2P Mesh	SFU	MCU
Operation Server Cost		–	↑
Bandwidth	↑	–	↓
CPU server-side load		↓	↑
CPU client-side load	↑	–	↓
Latency	↓	–	↑

2.1.3 Scalable Media Server deployment

The media server has characteristics that need to be taken into consideration for its deployment. As stated previously, media servers started migrating to the SFU topology as

scalability is better to achieve. Video conference systems use many techniques to achieve scalability. Those techniques focus on specific scenarios that are common in video conference systems.

Horizontal Scalability

Recently, as video conference platforms gained popularity, many conferences or small meetings were happening at the same time. The architecture employed by those systems would need to provide answers to this increase in the number of meetings. Horizontal scalability of the system and most notoriously of the media servers was the technique that ensured that every new call would have an available media server to connect to. A media server has limited computational resources and once it is overloaded with many meetings, it is not possible to ensure the quality of the call or even the delivery of the packets. At this point, there needs to be another media server that is capable of hosting the call, or else the clients will not be able to connect. For that, a new media server instance is created and becomes available to host more meetings. The cloud makes this type of scalability very easy and efficient. As the number of meetings increases, more servers are added to match that spike. To save resources and money, when the number of meetings goes down, some servers can be shut down.

The issue here is how to define whether or not the media server is being overused. SFUs are not very CPU intensive because their objective is to forward traffic. The main bottleneck is commonly the bandwidth. It does not mean, however, that this is always going to be the issue. So, the definition of overused is unpredictable and hard to obtain. For this reason, it cannot be based on just a factor, like CPU or bandwidth usage, but needs to be an abstract value that tries to represent well the load of the media server. A service is responsible for calculating this value for each media server and creating a new instance when there is no media server available. Forwarding the clients to the most fitted media server is done using a load balancer. Amazon Web Services (AWS) services allow auto-scaling based on an arbitrary load value defined by the developer. The developer must find a good way of calculating this value to ensure that resources are being well utilized.

Metrics are essential for this scenario and need to be collected and available for the load balancer to choose the best media server. AWS collects metrics using AWS CloudWatch, providing hardware usages such as CPU and bandwidth usage. The number of connections to the media server, the number of calls that it is hosting, and much more obtainable information are also valuable. Since there is no formula or standard for defining this abstract value, it is up to the developer to find the best way of calculating it.

Vertical Scalability

Ensuring that every meeting has a host to connect to is important and is achieved with horizontal scalability. Nevertheless, big conferences are also an issue for the media server. More and more participants connect to the media server consuming resources, which needs to keep on forwarding the data to each one of them.

Vertical scalability refers to the upgrade or downgrade of a server's resources. They adapt to the variation in the number of participants in the meetings that it is hosting. Vertical

scaling of a media server is not straightforward and has limitations. There is a limit to how much the server scales and scaling hardware usually has a lower return of performance per cost than adding a new weaker machine for the same price. For this reason, this type of scalability is not easily accessible and ends up being inferior in almost all scenarios when compared to horizontal scalability. Another issue, in most cloud providers at least, is that the server needs to be shut down before it has access to more resources.

Regardless, media servers have important characteristics that need to be taken into consideration when trying to scale them.

Firstly, it is essential to understand the difference between stateful and stateless components and what they imply in their deployment. In a stateless connection, the session state is shared among all nodes. A component can be removed without affecting the ongoing sessions. Whenever a new request is made there is no guarantee that the same machine is answering the request. Web servers are common examples of this stateless behavior. However, if a component has a stateful behavior, whenever the client connects to it, it will keep that connection open and communicate only with that machine. After connecting to the media server using the WebRTC protocol, the data is sent and received through that connection. This affects scalability because it means that the stateful component cannot be shut down while holding connections and the clients cannot make requests to a new server without instantiating another connection to that server.

A SFU maintains connections with all the clients in a group call since it needs to forward the traffic between them. However, if the number of participants in a group call increases to the point where the media server resources become insufficient, the call quality and reliability will go down. The server needs more resources, but as mentioned previously, to get more resources the server needs to be restarted, meaning that all the connections to the media server would be lost and the clients would need to reconnect, resulting in moments of downtime for all the group calls connected to that server.

Because it is not possible to predict the exact number of participants in a meeting, resources cannot be assigned apriori. Video conference platforms sometimes cap the maximum capacity of meetings to ensure that the media server does not need extra resources. It is much easier to balance the load with this approach. Large meetings and conferences are still available but are considered as a different type of group call, having access to better resources, and are treated differently in the load balancing algorithm.

Cascading SFU

Instead of having a single media server responsible for the entirety of a group call, spread the participants across other media servers and have the media servers establish a connection with each other. Cascading SFU[15] is a distributed approach to the problem of scaling a conference.

In theory, calls with unlimited size are achievable. In practice, it is not feasible due to the delay in communications that results from cascading. When a participant tries to join a group call but the responsible media server is not capable of handling another connection.

The participant connects to another media server instead and the media servers establish a connection between themselves. Note that each media server needs to have a reserved number of connections so that it can scale by connecting to another server. At least one reserved connection per hosted call is required for the system to work properly.

At a global scale, cascading SFU performs better than its competition[16]. One of the problems with spreading the media servers across the globe is that there is no realistic way of knowing from where clients connect. Some expectations are part of the thinking process when it comes to choosing the right server for a conference. As an example, if a user from North America is the first to join a meeting, then it is expected that the other participants are close, and so, a server in North America is chosen over the other options. But if the next two participants are from Australia, they will have to connect to a media server that is further away from them which delays the communication between two geographically close peers. In this scenario, the North American media server will need to send data through two distant links. More distance is usually related to more delay. An alternative would revolve around having the two Australian participants connect to a media server in Australia and have that media server connect to the one in North America. Figure 2.5 shows the example on a map for better understanding. With the addition of this media server in Australia, Australian participants can communicate faster, and more importantly, there is only one connection going around the world.

Figure 2.5: Example of a SFU cascading on a global scale. [16]

Cascading is not to be abused, each new hop results in a delay in communication. Adding unnecessary media servers not only slows down communications but also wastes resources and money.

Signaling Server

Another related component that needs to be deployed and scaled is the signaling server. This server helps to establish the WebRTC connections and although in most scenarios its tasks do not require many resources, it needs to be scaled to handle the increasing number

of participants. Requests are made to a single point, a load balancer, that distributes the load across multiple signaling servers. Since their use is ephemeral and client states are discarded after the WebRTC connection to the media server is established, traditional horizontal scalability is sufficient in the majority of the use cases. Ensuring that the clients interact with the closest signaling server can induce a considerable improvement in the speed of communication.

2.1.4 Commercial video conferencing systems

There are many examples of enterprise video conference platforms. Most of them are close sourced making it hard to study them. This subsection makes Zoom and Discord the objects of study, two successful video conference platforms with different use cases and goals. Zoom is a professional tool and is preferred for formal and business-oriented meetings. Other available platforms for the same purpose are Microsoft Teams, Webex, and Jitsi meet. On the other hand, Discord is geared towards gamers, providing features like voice channels, file sharing, and a more personalized experience for the application users.

Zoom

The ability to scale on demand is a hot topic in software architecture. Platforms need to be able to respond to a possibly high number of concurrent users. Zoom is one of the best examples of platforms that suffered from a sudden increase in traffic. Zoom's architecture and the components involved within the connection flow are an important case of study for someone who intends to build its video conference platform [17], [18].

Zoom resorts to AWS and Oracle servers to handle non-meeting-related tasks, such as scheduling calls and managing participants.

It typically relies on its data centers to handle video calls. Yet, if the demand for the service becomes too high and the data centers become overwhelmed, Zoom may transfer some of the video calls to AWS or Oracle servers to manage the load and maintain the quality of the service[19].

This strategy allows them to scale on demand and helps them ensure that the services provided maintain high quality even when there is a lot of demand. To minimize delays in communication, Zoom distributes its servers around the world. P2P connections are used when there are two participants in a meeting, providing fast and reliable communications[20].

Figure 2.6 presents a high-level Zoom architecture.

A typical Zoom Data Center is made of Meeting Zones, logical associations of servers located in a single global data center or inside an association network.

The Zoom Zone Controller manages, and load balances the requests made to that zone, forwarding the requests to the media servers. Load balancing of the requests dictates the load in the media server which, if done properly, improves the system's performance. This component is essential to ensure the scalability of the system.

The Multi Media Router(MMR) is the component that distributes the audio and video content between all participants. It can be seen as a media server with a distributed SFU

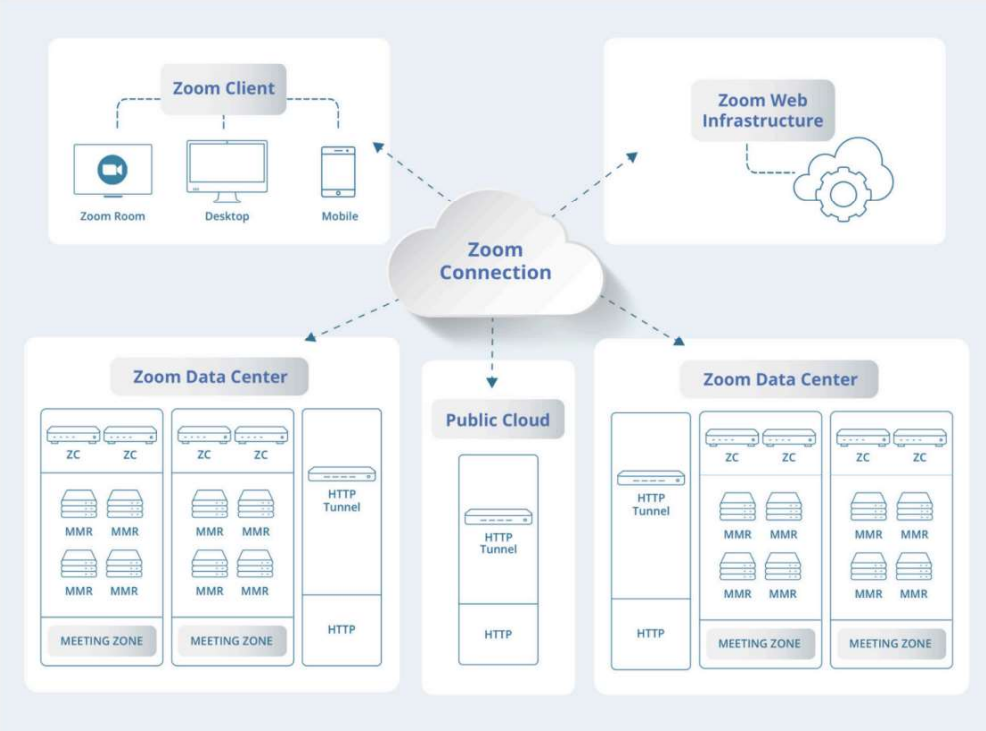

Figure 2.6: High-level Zoom architecture diagram.[17]

architecture [21], [22]. The optimization techniques employed by this component enable Zoom calls to handle large amounts of participants.

Zoom collects metrics about the CPU and bandwidth usage of the clients to understand the data quality that best fits. Multi-bitrate encoding enables this by having a single stream containing every resolution needed[23].

In summary, Zoom uses a distributed SFU architecture globally distributed across data centers. To maintain the quality of all the calls, each data center has multiple zones. The zone controller load balances the requests that arrive at that zone and distributes the load between all the available servers in that zone according to the load balancing strategy. Many optimizations are done in the media server to improve the performance and reliability of the calls.

Discord

The solution provided by Discord aims at solving a different problem. Instead of scaling each conference to handle a huge number of participants, Discord needs to withstand a great number of small meetings. Their client is available as an application and as a webpage. The application offers more functionalities and a better user experience.

Users connect using WebRTC and the signaling is made using WebSockets. There are three important components in their architecture responsible for assigning and establishing communications, Discord Gateway, Discord Guilds, and Discord Voice[24].

Upon entering the application, the client establishes a WebSocket connection with a Discord Gateway server. This connection provides the client with events.

Discord Voice servers contain two components: a signaling server and a media server. They use a homegrown SFU server with various optimizations to video and audio for better user experience and voice call scalability.

A guild represents an isolated collection of users and channels. The Discord Guild server watches the server discovery service and assigns the voice server with the lowest load in a given region to a guild.

These three services are deployed in Google Cloud Platform all over the world and are all horizontally scalable. Discovery services handle server failover by continuously monitoring and assigning users to a different server whenever there is a failure.

2.1.5 Kurento

General-purpose open-source solutions started to appear as WebRTC technology gained more traction. Kurento[1][25]–[27] is a WebRTC Media Server that offers capabilities such as group communications, recording, routing, transcoding, and mixing. It differentiates itself from other open-source media servers by having a modular architecture that allows developers to expand its capabilities with features such as computer vision, augmented reality, or speech analysis. An API in Java and Javascript is available for developers to interact with the low-level media server, permitting a more enjoyable developing experience. Two important concepts in Kurento serve as a building block for developers. The Media Element works as a generic unit that performs a specific action in a media stream. This element abstracts developers from the low-level operations over the stream that happens in the Kurento Media Server(KMS). They receive media from other media sources and can send it through media sinks. Media Pipelines function as a connector between Media Elements, linking the output(sink) of an element to the input(source) of the following. The conjugation of Media Pipelines and the prebuilt Media Elements that Kurento offers allows for a variety of use cases. Media recorders, image filters, and media dispatchers are among some of the many available media elements.

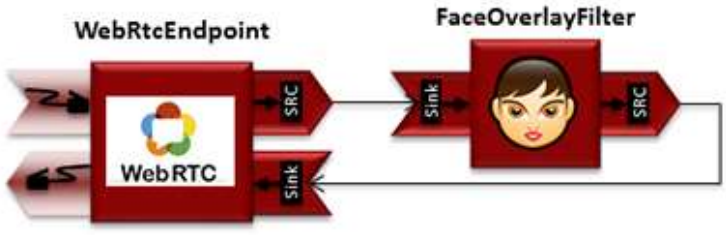

Figure 2.7: Example of a Media Pipeline implementing an interactive multimedia application receiving media from a WebRtcEndpoint, overlaying an image on the detected faces and sending back the resulting stream.[25]

[1]Kurento: `https://doc-kurento.readthedocs.io/en/latest/index.html`

On that note, Kurento provides templates for the developer to create its personalized media element. Kurento supports customized GStreamer[2] and OpenCV[3] modules. GStreamer confers low-level control over the video streams. Meanwhile, the OpenCV module, built on top of GStreamer, is best suited for manipulating the images obtained from the videos.

As for the media server architecture, Kurento supports both the MCU and SFU architectures. The developer is responsible for establishing the connections between peers.

The creators of Kurento also worked on the project NUBOMEDIA[28], a Platform as a Service(PaaS) to simplify the development, deployment, and scalability of WebRTC applications.

2.2 Emotion Recognition

The integration of machine learning models requires knowledge of how they operate, most notably for our use case, what the models use as input, and what operations are being made to the audio and video data. This section aims to briefly explain how the process of emotion recognition is being executed by the state of the art methods. Two emotion recognition categories are of interest for this work: Speech Emotion Recognition (SER) and Facial Emotion Recognition (FER). The first one operates on audio signals, while the second operates on video.

2.2.1 Speech Emotion Recognition(SER)

The emotion of the speaker can be understood through the characteristics of the voice. Being able to understand the emotion of the speaker is very important as it can influence the flow of the conversation. There are many real-time scenarios where SER can be applied. E-learning, call centers, and lie detectors are just a few examples.

The state-of-the-art approaches for SER can be divided into two groups: Deep learning and machine learning. Deep learning approaches use powerful models and are usually slower. Machine learning is the term given to more simple models that do not have many layers. The review article [29] presents a study of the current trends for SER.

In the Convolutional Neural Network(CNN) models, the audio input is transformed into an image format, in most cases a variation of a spectrogram. Newer approaches took one step further. For the CNN to make predictions on the audio signal it needs to have a fixed size, so the audio is split into chunks. However, CNNs are context-free, meaning that they make predictions disregarding previous inputs. Newer approaches are using other models that can understand the context to support the CNN predictions[30]–[32].

Non-deep learning approaches operate in a smaller set of features. Instead of using all the features of the audio, there is an attempt to find the most useful features. For instance, mel-frequency cepstral coefficients are a representation of the audio that reduces the dimensionality of data by including only the audio features that the human auditory system can process. Other approaches might use their own set of features retrieved from the audio signal, whether

[2]GStreamer: `https://gstreamer.freedesktop.org/`
[3]OpenCV: `https://opencv.org/`

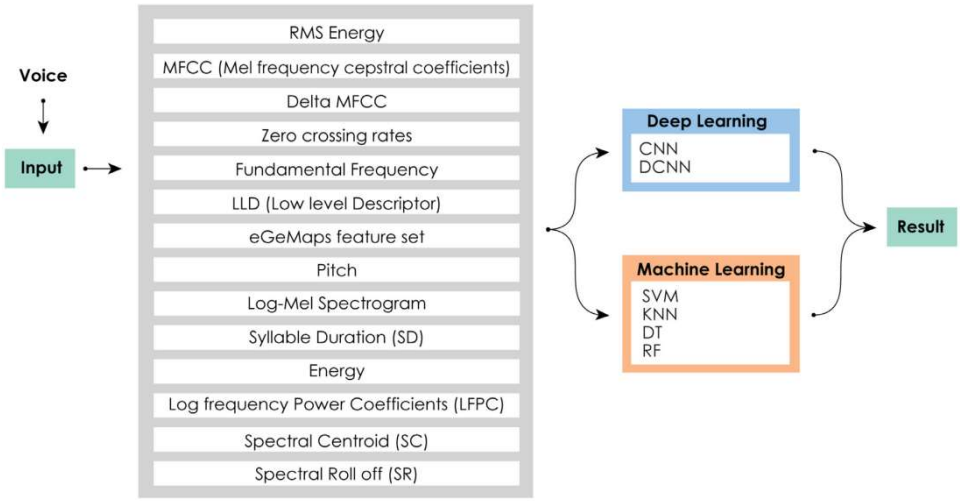

Figure 2.8: State of the art approaches for SER.[29]

by selecting them by hand or by using a machine learning model. This will reduce the dimensionality of the audio while trying to maximize the retained information. The selected features are then used to train simpler models such as random forest or SVM[33], [34].

In summary, audio signals are processed in chunks. In deep learning approaches it is being explored the ability to add context to the models. In machine learning approaches, it is used a subset of the audio chunk features as the input for the model which usually results in faster classification.

2.2.2 Facial Emotion Recognition(FER)

Several steps are required for Facial Emotion Recognition(FER). It starts with the face detection step, where it is used a face detection model like Haar Cascade detects the face in an image[35]. Facial features are retrieved from the eyes, lips, mouth, and anything else that changes in the face when an emotion is being displayed by the person. Researchers would select these features by hand but nowadays machine learning classifiers are the ones selecting the features. Similarly to SER, there are two groups of models: deep learning and machine learning models.

Deep learning research contains similar approaches. Models are trying to add context to CNN based models by utilizing other models such as LSTM[36]. Transfer Learning is also an alternative by using pre-trained large networks[37]. The capture and selection of facial features are still being explored with many different approaches being tested. In general, most approaches use a CNN based model but try to add context by combining it with an LSTM.

Machine learning approaches use handcrafted feature extraction techniques. These include extracting facial landmarks such as the position of the eyes and nose. Some techniques extract features on a temporal level by capturing changes over time. Many approaches use dimensionality reduction techniques such as Principal component analysis(PCA) and the features are fed into SVM or CNN based classifiers.

In summary, models can operate per frame, analyzing each frame as they arrive. They can also operate on multiple frames to examine the changes that happen over time.

2.3 Machine learning for video conference data analysis

2.3.1 Machine learning applied to video conference data

The increase in demand for video conference platforms created space for a new set of functionalities that were under-explored. The amount of data generated can be used in many machine-learning problems, such as image and audio classification. Emotion recognition is a subject of study to develop algorithms and find techniques capable of recognizing emotions based on human feedback. This feedback is obtained from face gestures as well as voice patterns. Video conferences can capture those features from the generated video and audio.

Meetings and conferences can be recorded for posterior analysis, producing a report that becomes available for later user observation. This process can be referred to as offline analysis. Generally, this type of analysis is well-detailed as no time constraints are associated with it. For emotion recognition, having no time constraint allows the use of powerful machine learning algorithms that provide more detailed and accurate information. Data is processed in batches and distributed across the available computational resources[38]. Tools like Apache Spark[4] are great options for this type of workflow[39].

On that note, the algorithms do not need to be used just for offline analysis but can be exposed as features for the platform users. Users might need a live video and audio analysis that provides real-time feedback. The video conference platform exposes this feature to the clients, allowing them to activate it and receive feedback about the analysis. Real-time analysis poses a computational challenge. The machine learning models need to be applied to the continuous data and return the results fast enough so that the client's experience is not ruined. Some of the solutions have a tradeoff between the speed and accuracy of the model. In typical scenarios, it is observed that the live analysis is less accurate since it needs to be faster. In contrast, the offline analysis can compensate for that by having a more detailed analysis.

The system needs to distribute the data that is provided by streams. Apache Spark has support for streams and is one of the possible tools for this scenario[40]–[42]. Machine learning frameworks like TensorFlow[5] and PyTorch[6] offer serving capabilities. TensorFlow Serving and TorchServe do not support streaming by default but can be integrated with stream processing tools like Apache Flink[7][42].

Finding the optimal tradeoff where the prediction time is sufficient and the model accuracy is acceptable is challenging[43]. It starts by understanding the available hardware for inference. For example, not having access to powerful GPUs will most likely be infeasible for deep learning models. Then it is essential to remember that inference time is only a partition of

[4]Apache Spark: `https://spark.apache.org/`
[5]TensorFlow: `https://www.tensorflow.org/`
[6]PyTorch: `https://pytorch.org/`
[7]Apache Flink: `https://flink.apache.org/`

the waiting time. Communication between clients and the media server, communication with architectural components, and processing times are altogether an issue. Inference time can be reduced by having better hardware or a faster model. Ensuring that the inference components are geographically close to the client increases the communication speed. The choice of the model is essential for this use case and needs collaborative work between the data science and operations teams.

2.3.2 Machine Learning Operations

Part of this work focuses on the deployment, maintenance, and monitoring of machine learning models. This falls into the area of Machine Learning Operations. MLOps refers to a collection of methods and approaches that strive to enhance the cooperation, communication, and coordination between data scientists and operations teams during the creation, deployment, and upkeep of machine learning models. There is a set of principles and guidelines required to realize MLOps.

According to Kreuzberger *et al.* [44], there are nine important components used in MLOps.

The automation of processes using CI/CD enables seamless integration, delivery, and deployment of changes. The automation of the building, testing, release, and implementation phases, provides the developers with prompt feedback on the outcome of specific steps which in turn increases the overall team efficiency[45]–[47]. GitHub actions, GitLab, and Jenkins are common technologies used for this purpose.

Source code repositories provide collaboration and communication to the different elements of the team. Each new change to the product is stored and labeled, guaranteeing the versioning of the project[46], [48]. Versioning ensures the traceability of errors and the reproducibility of previous tests. Tools such as Github, Gitlab, and BitBucket are great source code repositories.

The machine learning workflow is composed of many steps that can be represented in a directed acyclic graph(Dag). Workflow orchestration defines these graphs specifying the order of execution of each task by taking into account any relationships and dependencies. This way, the machine learning workflow is repetitive, automatic, and easily reproducible. There are tools available for building the graphs with the pipeline workflow. AWS SageMaker Pipelines and Azure Pipelines are among the most popular ones together with Luigi and Apache Airflow.

Feature stores are a recent concept and work similarly to data warehouses. They are a central repository that stores the most commonly used features and serves them across machine learning models and teams[49]. Data scientists do not need to process the features for every new model and it helps maintain consistency in the models and documentation across all the machine learning projects within the organization. Although this type of storage can be replaced with any kind of data store, it provides versioning and reproducibility while enhancing the collaboration between members of the team.

The complexity of many machine learning models results in a great training time. The need for a model training infrastructure that speeds up the training process is essential to ensure better results[45]. This infrastructure needs to be scalable and distributed provided

that the individual machines have access to powerful hardware and notoriously powerful GPUs for deep learning models. Local solutions are possible but require a great investment[50]. On the other hand, cloud computation gives access to a more scalable and distributed solution, making it easier to obtain more computation resources and balance the load between them. Cloud platforms such as AWS and Google Cloud have services that facilitate the scalability and load balancing of the system.

Trained models are stored in the model registry[45], [46], [48]. This can be seen as normal storage that maintains the different versions of models. It is important to ensure the reproducibility and versioning of the models. Some tools specialize in storing models such as AWS SageMaker Model Registry and MLFlow but other types of storage like AWS S3 are sufficient.

Another used component is ML metadata stores[49]. During the machine learning workflow, each step generates metadata. For example, the model training phase includes metadata regarding hyperparameters and evaluation metrics, in the initial phase the metadata can be the name and version of the dataset. Metadata stores facilitate the monitoring, comparison, organization, and filtering of metadata.

To make the model available for production, it needs to be deployed taking into consideration the requirements of the product. The model serving component is of extreme importance because it will reflect the user experience of the clients. Preferably it should be scalable and distributed to respond to requests with minimum delay. Trained models are usually containerized using technologies such as Docker while being served through an API. This component is explained in detail in the next subsections.

The last component referred to in the paper is responsible for the monitoring of the model serving component and the ML infrastructure[46]. Tools for this component include Prometheus with Grafana[51] or in case of using the AWS infrastructure, AWS cloud watch, and AWS SageMaker model monitor.

Figure 2.9 shows an example of a MLOps pipeline depicting the components and their place in the pipeline.

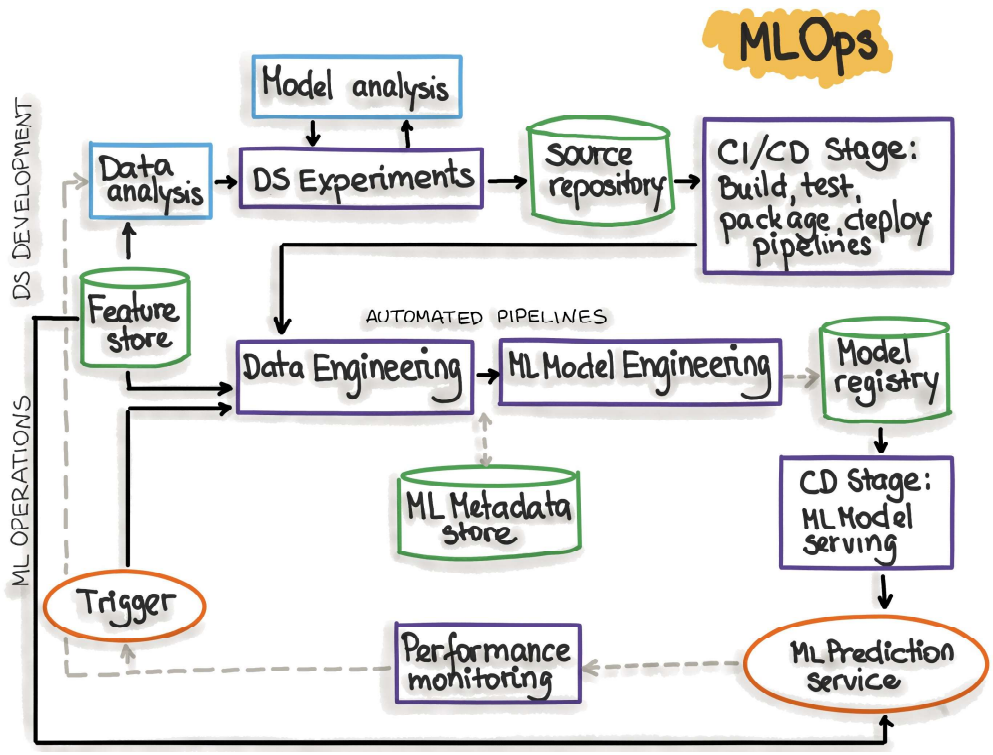

Figure 2.9: Example of a MLOps pipeline describing the components and their position in the pipeline.[52]

2.3.3 Model Deployment

The machine learning lifecycle has more than just the model training and testing. There are many steps required for the model to be ready and available for clients to use.

It needs to be integrated into the right environment for it to be available to handle prediction requests. The model deployment step comprises multiple architectural decisions that enable it to answer the needed requirements. Scalability and monitoring are among those requirements. Those are the main objectives of the model deployment and model serving steps. To achieve said goals, there are a few commonly used strategies. There isn't a defacto strategy that is proven to be better in every scenario, leaving the choice depending on the product's requirements. The deployment is highly recommended to be automatic, without human intervention, saving time and reducing errors.

As previously mentioned, there are two scenarios for our use case, offline and real-time analysis. The latter poses a bigger challenge for model serving due to the necessity of a fast and yet accurate report.

Containerization

Ensuring that the deployment is made using containers is essential. They offer a predictable and controlled environment while being easy to scale and modify[46], [53]. Versioning becomes considerably easier, allowing newer and better models to replace older ones with fewer complications.

Monitoring

The continuous maintenance and monitoring of the model need to be assured. This way models are kept in the best conditions allowing the detection of possible rectifications that can be made in the model. Predictions and the received input can be stored for further analysis, providing data scientists with possible useful information that can guide them on how to improve the model. Prediction times, on the other hand, is a measurement that is useful for understanding the clients' experience as bigger prediction times result in the delivery of out-of-date reports.

Real-time analysis

The deployment architecture is responsible for the fulfillment of requirements such as performance. For real-time analysis, the architecture needs to effectively handle the performance requirement, otherwise, user experience and the application usability can be compromised.

Current state-of-the-art solutions revolve around having multiple instances of the model scattered across the globe responding to requests. Models can be deployed on a local data center or in the cloud using cloud provider services. Local data centers require an already-built infrastructure that might come with heavy upfront costs. Cloud providers are more accessible to the general and require less upfront costs while being easier to scale.

The machine learning model is served by a web server that receives the live data and returns the predictions. Data comes in the format of either a stream or can be already divided into chunks. As per usual, when it comes to horizontal scalability, the load balancing of the requests is essential to retrieve as much performance from the deployed instances. Scaling instances can take some time, and for that reason, there can be a drop in performance as an insufficient number of machines have to handle the excessive amount of requests until the other machines become available.

Serverless Computing

In recent years, serverless computing started gaining some traction as an alternative for machine learning model serving.

Functions-as-a-Service(FaaS) is a more explanatory name for serverless computing. AWS Lambda was the first service that implemented this concept and is still up to this day the most used among cloud providers. Instead of deploying code in a machine that is forever listening to requests, in serverless programming, the developer only needs to deploy a function with the desired functionalities. The function is called upon an event, receiving the necessary input data for it to work. While there are no events, the deployment is not consuming resources and thus saving money. For machine learning predictions, the function will receive the input data and execute the prediction using the trained model.

Although the concept of serverless programming might seem better for the machine learning model serving use case, it is a recent technology and thus has some problems that need to be taken into consideration before using it. Nevertheless, every scenario is different

and should be evaluated according to the needs of the product. Serverless has clear advantages and disadvantages over traditional deployments[54], [55].

The scalability of the model is assured by the service. As the number of requests grows, the service creates more instances of the function to handle incoming requests, meaning that the developer does not need to worry about scaling. Billing can also be attractive as, for example in AWS Lambda, the service works as a pay-per-use. Machines are not permanently running wasting resources and are instead ephemeral and discarded after use.

At a glance, this concept might seem a good idea but as it stands now, there are too many limitations[54], [55]. AWS Lambda does not offer a way of specifying the hardware where the functions are executed. For deep learning models that need a GPU to accelerate the predictions, this limitation is almost detrimental to the utilization of the service. The ephemeral nature of Lambda adds to the necessity of the system to load everything on startup, leading to slower responses. This scenario is described as a cold start. The added delay discourages the use of Lambda for real-time computing, where performance is essential.

Due to the mentioned limitations, model deployment in general-purpose servers is still seen as the best choice for real-time analysis for deep learning serving.

CHAPTER 3

Requirements Analysis

3.1 System concept

In a digitally connected world, remote collaboration has been greatly improved by tools such as video conference systems. Their effectiveness can be greatly enhanced by incorporating advanced technologies that go beyond the transmission of audio and video. The integration of machine learning algorithms capable of analyzing the users' emotions gives those users the ability to understand and respond to the emotions of their counterparts.

The creation of a robust and flexible pipeline ensures that users have access to newly updated machine learning models, capable of better understanding the participants' emotions. The system can maximize its accuracy by utilizing both the audio and video provided by the user, and as such, the pipeline needs to support both types of data. Since the machine learning pipeline needs to analyze live video and audio data, it needs to minimize the latency and delay of the responses.

The privacy of the data and the ethical implications of a video conference system are challenges that we are aware of. These are valid concerns for systems that work with the users' confidential data. However, these problems are not part of the objectives of this proof of concept system. Nevertheless, the construction of the system does not prevent these issues from being tackled in the future.

The integration of audio and video emotion recognition in a video conference platform can benefit different scenarios. In the case of e-learning, where the speaker is teaching or giving a lecture, obtaining feedback about the listeners' emotions can help them adapt accordingly. In a call center, real-time emotion analysis provides useful feedback that can lead to better interaction between the participants. In telemedicine, emotion recognition can assess the emotional state of patients, providing better care to the patients. There are other scenarios where this technology can be applied to people with disabilities, where the contents of the application can change according to the emotional feedback. These are high-level scenarios that depend on how the video conference application is used.

This system can be seen as a service that could be used by other systems. A video conference platform can use this system to extend its functionalities and provide real-time emotion analysis. We will be basing the system's use cases on a few functionalities that are observed in the target video conference platform we are aiming to support.

The target functionalities are simple functionalities of a group meeting.

- Distinction between participants - this is a simple scenario where the system has to operate knowing which participant is being analyzed.
- Distinction between rooms/groups - the system should be able to make the distinction between participants in different rooms.
- Audio and Video support - the system should be able to analyzed both audio and video separately, supporting the scenarios where one of them can be disabled.

Video conference systems can be used in many scenarios and have plenty of use cases, but this work focuses on a specific one. In a meeting, the speaker or the presenter can have an easier time adjusting their presentation and speech based on the emotional feedback returned by the system.

3.2.1 Functional requirements

Functional requirements play a pivotal role in defining the capabilities and behaviors expected of the system. In our system, there are a few key functionalities that need to be available for the user.

- Users connect to the video conference platform through the web and join a room.
- Participants have their audio and video analyzed by the system.
- The system should be able to use different classifiers for emotion recognition on both audio and video.
- System operations should be monitored offering quantitative evidence of performance and availability.
- The classifiers should be scaled dynamically according to the load.
- The system should be able to integrate with video conference platforms that use the Kurento media server.

3.2.2 Non-Functional requirements

Non-functional requirements describe how the system should behave, rather than the specific functions it should provide.

- Low Latency - the system should provide near real-time classification of the videos, with average responses no longer than 2 seconds.
- High Availability - the system needs to ensure minimal downtimes.
- Extensibility - different machine learning components need to be easy to integrate with the system.
- Scalability - the system should be able to grow according to the demand.
- Fault Tolerance - consistent monitoring of the system to detect faults.

3.2.3 Constraints

There are a few constraints related to this work.

The video conference platform that will used as a proof of concept has to use Kurento as a media server. This is mostly due to ongoing and previous works done by our research that also use this media server.

The system has to integrate machine learning models that were developed by the research team. However, it is not limited to those models, other models can be added.

CHAPTER 4

Design and Implementation

This chapter introduces the design and implementation details of the system. It offers an overview of the pipeline elements and their functions. As we delve deeper, we explore how these elements were conceived and the decisions taken during their development. The chapter outlines the general component architecture, elucidates their roles and implementation particulars, covers local deployment, and addresses pipeline deployment within a cluster setting.

4.1 General Overview

In the early stages of the architecture design process, the main goal was to identify modules of the system based on their unique purpose.

Figure 4.1 presents the identified modules. The connections between them are based on how the data is expected to flow.

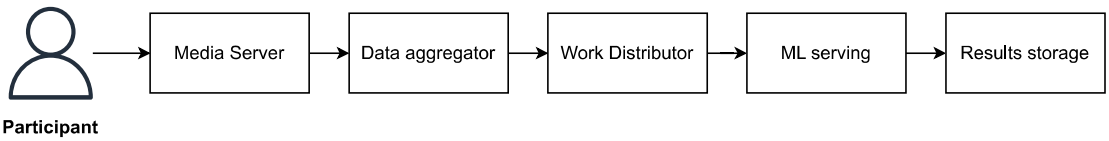

Figure 4.1: System modules diagram

The input data is provided by the client when it connects to a media server. The media server module contains all the logic of a video conference system. Clients connect using the WebRTC protocol and send audio and video data to the media server.

The data aggregator module is responsible for retrieving the audio and video data streams from the media server. It then aggregates the data in chunks and makes the necessary processing to ready it to be used by the machine learning models.

The work distributor takes the data chunks from the aggregator and selects a model to send the data for prediction. It needs to support scenarios where more models are added to the system.

The ML serving module exposes a machine learning model that can receive input and forward the output of the prediction. This module needs to scale based on the load.

Lastly, prediction results are stored in a storage component for later analysis and are sent to the client.

4.2 Components Implementation

4.2.1 System Overview

A system architecture was conceived based on the identified modules. Figure 4.2 illustrates the architecture overview of the system. The diagram displays two distinct instances of Kafka to enhance diagram perception and message flow visualization (in the actual implementation, only a single Kafka instance is utilized.)

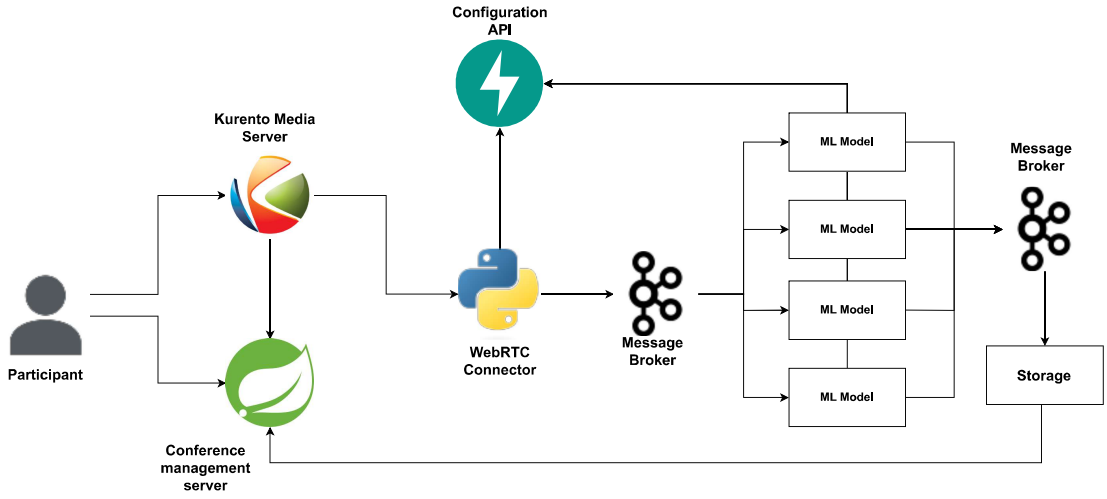

Figure 4.2: Architecture Overview

The illustrated architecture is composed of multiple components.

The **media server** module encompasses the **Kurento Media Server** and the **Conference Management Server**. The Conference management server is built with Spring and is responsible for enabling the user to connect to the media server. It serves as a signaling server and handles the logic of the video conference application such as video conferencing rooms. KMS is the media server and its job is to forward the media to the users of the platform. The word Kurento will be used to refer to both of these components. This module also has a web page from where the users can connect to rooms, talk with each other, and have very basic functionalities in a video conferencing system.

The **WebRTC connector** makes the **data aggregator** module. It uses a Python library called aiortc[1] that provides ways of establishing WebRTC connections. In this case, it is used to connect to the media server and extract the audio and video data of each user. It also has the objective of aggregating data in audio segments and video frames.

[1]AioRTC: `https://github.com/aiortc/aiortc`

Kafka covers the **work distributor** module. It is a message broker that is suitable for real-time stream data. Using Kafka load balancing and work distributing capabilities, the system distributes the frames and audio segments among the various machine learning models.

The **ML serving** module is composed of machine learning models that receive data from Kafka. They are constantly fetching messages that are the input for the predictions.

The **storage** module is very basic in our scenario. It receives messages from Kafka containing the results of the predictions and stores them in the file system. This is just a simplified approach that enables us to observe the results without adding more complexity to the work.

To improve the interaction between the various decoupled modules, the system makes use of two components: **Kafka** and the configuration API. Kafka not only distributes the messages through the ML models but also serves the purpose of a message broker, distributing the messages through other components in the system. The configuration API is a web server running FastAPI[2] that provides the configurations of the models to the other components, mainly the WebRTC connector.

However, due to encountered implementation issues detailed in the subsequent section, the actual architecture differed from the initial depiction. Challenges arose in establishing a connection between the WebRTC connector and the KMS, unsolvable with our best efforts. Consequently, an alternative blueprint was conceived to overcome this constraint. The workaround solution is visualized in Figure 4.3.

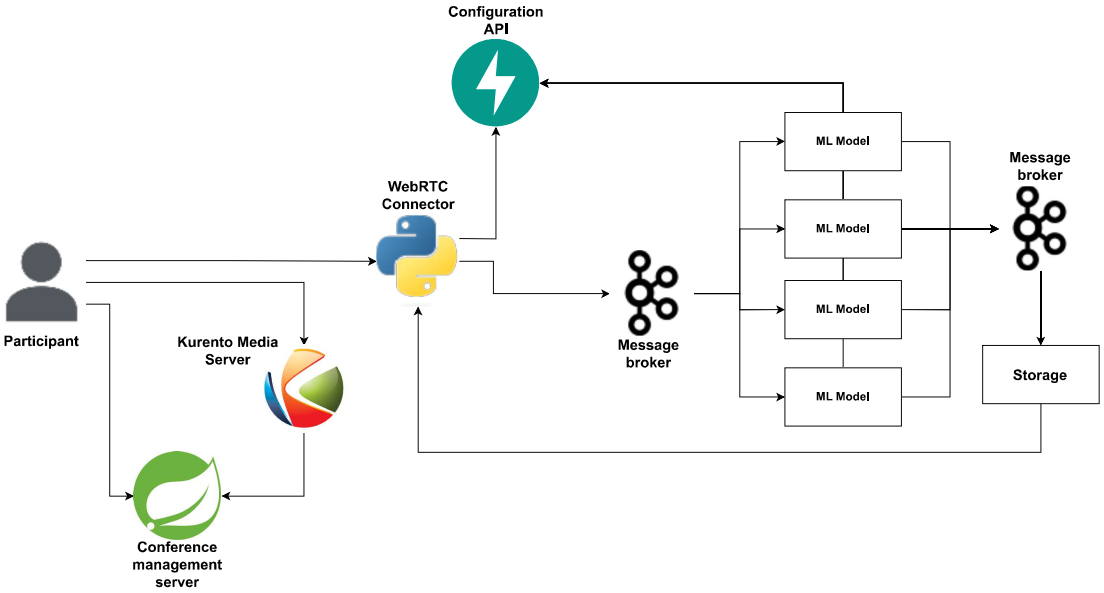

Figure 4.3: Alternative Architecture Overview

In this revised approach, the WebRTC connector bypasses connecting to Kurento and instead establishes a direct link with the client. This configuration liberates Kurento from

[2]FastAPI: `https://fastapi.tiangolo.com/`

the other parts of the system, assigning it the singular task of video conference management, receiving and relaying media between clients.

4.2.2 Kurento and video conference application

Kurento is an open-source project suitable for building modular WebRTC applications. Its documentation features numerous illustrative samples, adaptable for a variety of goals. The chosen sample for this project was the Group Call example that provides a web interface and the essential Kurento logic for multi-participant video conferences, supporting the creation of multiple rooms. This example aligns with the project's requirements, requiring minimal code alterations.

Setting up the example requires a few components. Initially, the Kurento media server must be running, achievable via the supplied Docker image in Kurento's documentation. It operates on port 8888 and exposes an API to control the low-level media elements. The Spring web server that is provided in the example, makes use of that API to manage the participants and coordinate the conference rooms. A simple web interface is exposed the the participants to join the rooms and communicate with each other.

The simplified web page features a form soliciting the user's name and desired room for admission. There are no constraints on the provided name and no integrated login logic, as those fall beyond the project's scope.

The required messages for establishing the WebRTC connection are exchanged upon completing and submitting the form. Here, the Java Spring application functions as a signaling server, intermediating between the client's browser and the media server. Websockets were employed for message delivery between the participant and the web server. If all proceeds smoothly, the WebRTC connection is established, granting the user access to their camera feed and those of fellow participants in the group call.

Figure 4.4 presents an example of a group call with three fake users.

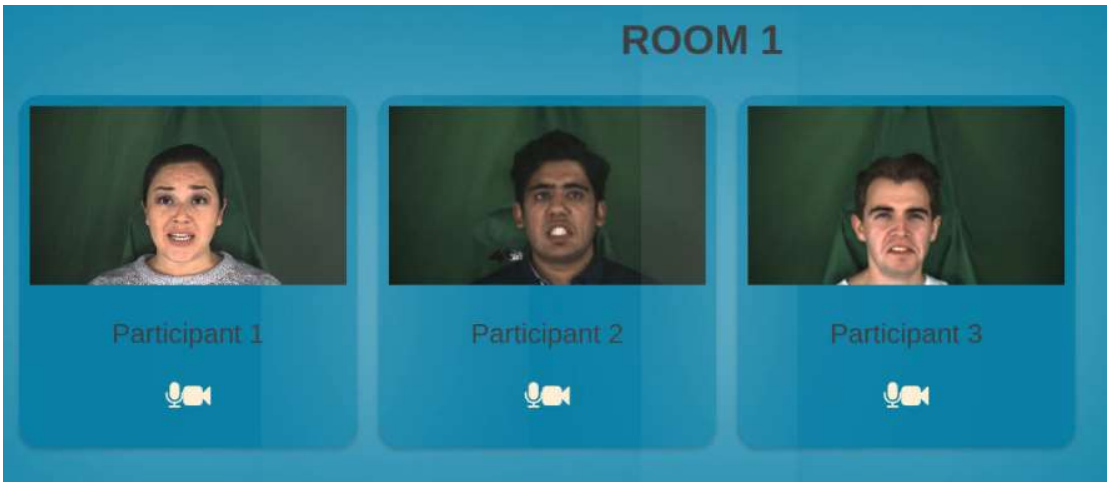

Figure 4.4: Webpage interface example with three mock users in the same room.

4.2.3 Model storage and version tracking

The integration of machine learning models is the focus of the pipeline. Two common steps were identified while using a model: the model initialization and the prediction of input data. Since the models followed this behavior, a Python script was created that focused on connecting the pipeline with the developed models. This script is responsible for initializing the model and for forwarding the messages from Kafka to the model for analysis. Python was used since the available models were also built using Python.

This script was built upon some expectations that need to be met for it to work as expected. It requires a list of files that vary from model to model. At first, a Python script with the name classifier.py must be present, containing a class named Classifier and having two methods: the init method and the predict. This way it is possible to ensure that, independently from the model that is used, if they follow these naming conventions, it is possible to load the model and send data for prediction. Other files such as the Python requirements, operating system libraries, the model configuration file, and the model parameters must also be present.

With a base script that initializes the model and consumes messages from Kafka, it was possible to serve different models by just changing the required files. It then becomes essential to keep track of existing models and their versions to facilitate accessibility and reusability. Model storage is one important concept in MLOps and is achievable in many ways. This concept promotes the reusability of developed models by storing their state after training in a shared storage and enables their usage outside the experimental environment.

Having access to the models' versions and an easy way of experimenting with them can be essential for speeding up the work of the data science teams. This creates the necessity of having a shared storage where each model version is identified accordingly so that in the future it is possible to differentiate, analyze, and possibly rollback between them.

The proposed solution for model storage with high availability and shared access relies on GitHub Packages, a model registry for storing container images. The presence of the codebase in GitHub impacted the choice of this particular registry.

At first, a base docker image was created containing the shared code between all the existing models. In this case, the image contains the common script that initializes the model and consumes Kafka messages. Every model will have its image using this one as the base image.

Naturally, the constituents of the model were assembled into a single docker image to promote reusability. When the image is built, it installs the Python and operating system libraries listed in the provided files. When it comes to versioning, most image registries, such as GitHub Packages, already provide versioning of the stored images by referring to them by name and tag. Thus, the image name and tag correspond to the model's name and version, respectively. This approach is more maintainable and simplifies and helps automate the deployment of the model in the pipeline.

Figure 4.5 provides an overview of a generic model image and its contents.

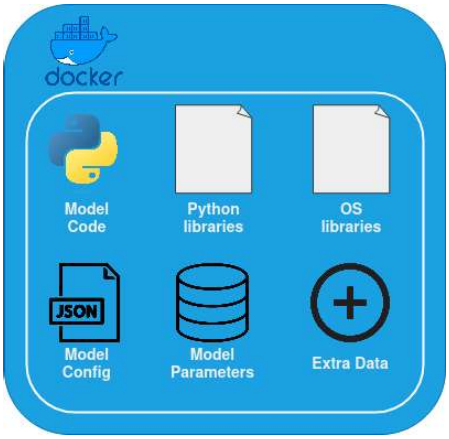

Figure 4.5: Common structure for containers(Docker) providing machine learning models compatible with the pipeline.

To optimize and ease the process of creating this image, a script was written that already creates these files. It is up to the model developer to change its contents to match the specifications of the model. The script also initiates a Dockerfile that is responsible for creating the resulting image.

4.2.4 Pipeline configuration

Different models expect the pipeline to work in a specific way that might be completely distinct from another. This might be due to the lack of standardized training data, model specifications, or the need for data to be processed differently. The creation of a pipeline that is capable of adapting to the needs of each model is essential for its testability and reusability.

This was achieved by providing the model's configurations to each of the components in the pipeline. A central component implemented as a web service is responsible for serving the configurations to the other components.

Models register their configurations as they are ready to serve. Every other component can obtain the configuration for a model with a simple request. This process requires that a configuration file is stored together with the model but makes the pipeline more flexible for newer developed models. These files are stored in a JSON format and contain information related to the way the individual components should handle the data.

In a general scenario the message flow to store and exchange configurations can be seen in figure 4.6. The WebRTC connector makes one request for each stream it processes (audio and video).

Configurations are identified by the correspondent model name and tag. For this reason, it is very important in this pipeline to ensure that models are correctly identified by their unique names. In the snippet of code 1 it is presented an example of a potential configuration for a model. The provided example shows a possible configuration for a classifier with the deepface that is used for image classification. It provides configurations that change the behavior of the connector. For example, it aggregates only 1 frame at a time and steps over 15 frames.

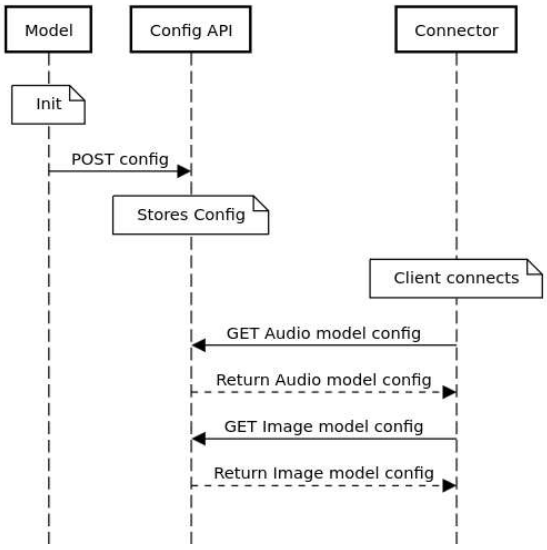

Figure 4.6: Configuration sequence diagram.

```
{
    "name": "deepface",
    "tag": "1",
    "type": "image",
    "connector": {
        "aggregate": "frames",
        "value":  "1",
        "step": "15"
    }
}
```

Code 1: Example of a model configuration file.

Configurations work as a proof of concept and there are not many supported configurations for the moment. The main objective for this feature is that it becomes much easier to integrate newer models that have new necessities.

It is an important step to improve the extensibility and modifiability of the code.

4.2.5 Data collection and aggregation

Video conferences create data streams to be processed, however, the developed models operate on a fixed input size. We explored the possibility of having a component capable of aggregating the data in chunks to be analyzed by the models. One of the aspects of this component is that it needs to be flexible by adapting to new models and their corresponding configurations. Some options were explored to understand which one would better fit our use case.

Kurento GStreamer Module

Initially, we consider using the extensibility features provided by the Kurento media server to prepare the chunks of video and audio in the correct size expected by the classification models. Kurento allows the creation of custom modules, however, that requires a good

knowledge of GStreamer, a low-level framework for handling media components that serves as a foundation for Kurento. Due to the limited time, a steep learning curve, and an overall lack of examples, this option was abandoned. There is no guarantee that this solution would work without further testing.

However, we derived some of the pros and cons of this solution based on a pure theoretical hypothesis. In terms of performance, it is evident that this solution imposes a higher load on the side of the media server as it needs to process and analyze the streams of data of each participant. Since it was not tested in practice, it is not possible to understand whether or not this would produce a noticeable delay in the video conference system. In terms of scalability, adding more processing to the media server decreases the number of participants that can connect. Another characteristic that was not adequate for our use case was the lack of flexibility in this approach. The integration of pre-existing machine learning models is complicated and would require modifications to the code for every newer model.

As for the advantages, the processing of streams in the media server itself results in fewer connections and hops between components. The travel time between components adds significant delay to real-time analysis. In some scenarios, the additional delay might be so pronounced that it would be better to make the analysis in the same component. This solution would be a better fit for when models are simple, fast, and do not cause a noticeable overhead in the media server.

Python Aiortc

Although WebRTC was initially proposed for web browsers, some implementations try to follow the Javascript API. Aiortc is an open-source library written in Python that provides ways of establishing a WebRTC connection.

Using this library it is possible to receive the data from the user's camera and microphone. Frames can be aggregated and go through some processing before being sent to the machine learning predictors. Receiving and handling the data frame by frame might be slower, especially considering that the code is written in Python, but in turn, provides much more flexibility in terms of operations that can be done to the stream.

The aiortc component would need to mimic a Javascript client and connect to Kurento, allowing Kurento to forward the data streams as it does with the other participants in the video conference. Attempts were made to establish the WebRTC connection between Kurento and the aiortc component but we were not successful. Due to the time constraints and the difficult debugging nature of WebRTC connections, the solution was simplified.

Instead of receiving the data streams from the media server, the aiortc Python client receives the data directly from the browser. This requires the clients to establish two connections: one with Kurento and another with the aiortc component. Data sending connections are computationally more intensive which might make this solution less viable for a production environment. For that reason, this solution is not the ideal one but can be seen as a workaround that can be improved.

The WebRTC connection between the client browser and the aiortc component is indepen-

dent of the Kurento one. The WebRTC connector runs a web server using Fastapi that permits the client to connect using WebSockets. The WebSocket connection is used to exchange the required messages to establish the WebRTC connection. Aiortc then provides easy access to the frames of the stream. Since the majority of machine learning models are built in Python using powerful libraries such as numpy[3], this approach integrates very well with the rest of the pipeline as frames can be converted to numpy arrays.

Figure 4.7 shows a simplified version of the interaction between the connector and the client.

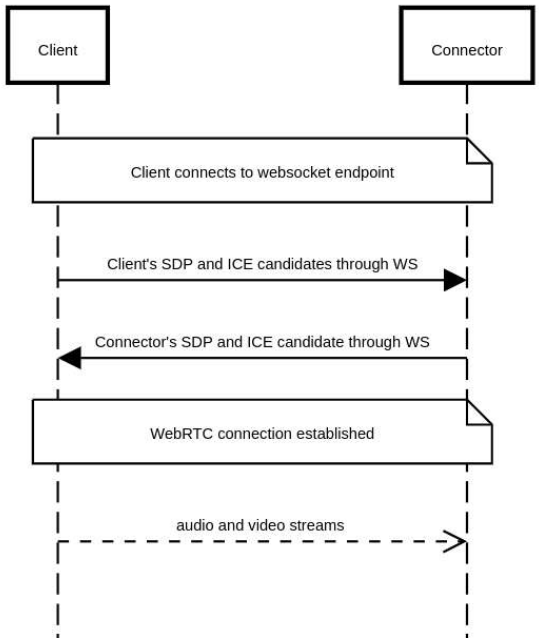

Figure 4.7: Simplified message exchange between the client and the WebRTC connector.

Also, components might need to adapt to the configurations of a given model. Upon establishing the WebRTC connection with the client, this component requests the configuration of an available model for that data type(audio or video). The received configuration specifies how to aggregate data, whether in frames or seconds, and the size of the chunks. The aggregated data is then sent to the next component, Kafka, which will be responsible for distributing them across the correct machine learning instances. For monitoring purposes, messages contain more than the client's audio or video frames. Model information and the initial timestamp of the aggregated frames are also stored in the message, providing useful information that can be used later for monitorization of the model's performance. The next subsection goes into more detail about the topic selection and message format.

[3]numpy: `https://numpy.org/`

4.2.6 Model serving

Kafka vs Web Servers

The most common approaches for serving machine learning models rely on a load balancer or a message broker to distribute the input messages across the running instances. Apache Kafka was selected to ingest the input data and distribute it across the running machine-learning model predictors. The choice of using Kafka was mostly based on the needs of our use case.

A valid approach can be built using web servers. These servers would receive the requests that were forwarded by a load balancer and make predictions for that input data. Adding a custom configuration to that load balancer creates a scenario where it is possible to have the same server analyzing all the frames of a participant. For models that require knowledge about previous frames, this approach can be very useful and powerful. Nevertheless, managing and configuring the load balancer might not be as easy which is why Kafka was chosen. It enables parallel processing using topic partitions and the distribution of messages is handled by Kafka itself.

One key aspect that is common in both approaches is that, to maximize performance and reduce the delay, the component that is receiving the streams and aggregating them cannot be stuck waiting for the prediction response. In the Kafka solution, this scenario is already handled as the aggregator component sends the message to Kafka and keeps on receiving frames from the streams. The web server solution needs to be adjusted so that the requests are waited asynchronously, possibly by using other threads.

Overall, both solutions are capable of handling this task but Kafka seemed to be the easiest one to make it work.

Work distribution

Kafka has the capability of distributing messages across multiple instances of the same consumer group, ensuring that every message is only sent to a single instance. This is essential to guarantee that there are no resources wasted on processing the same message.

Topic partitions allow Kafka to split the messages across multiple instances, enabling parallel processing. It is important to note that a partition will be assigned to a single consumer and thus the number of partitions needs to be at least equal to the number of consumers otherwise some of them will become idle. Kafka also tries to evenly split the number of partitions through its consumers, balancing out the load in the running instances.

Understanding those concepts, some strategies were designed to create a flexible and efficient pipeline. The next subsection describes the Kafka message workflow that enables the many components to exchange messages.

Firstly, in this pipeline, the concepts of consumer and topic can be represented by a model instance and the groups of instances of that given model, respectively. This way, for each model was created a topic where the instances running that model need to subscribe. By putting all those consumers in the same consumer group, there will not be any data duplication and thus there will not be reprocessing of the same message by another instance.

As for the number of partitions, it will depend on the number of instances that are required for the use case in question. This will be addressed in more detail in the Kubernetes section but assume that for local deployments that number is static and does not increase with the number of instances.

Model serving instances are stored inside a container and they make predictions on messages pooled from Kafka. In terms of flexibility, the pipeline automates the creation of the Kafka topics so that when a new model is deployed it will work without any human intervention. Upon instantiating a container of a model image, it registers its configuration in the configuration web service which in turn is responsible for creating the Kafka topic for that model if it does not exist yet. The WebRTC connector, after the creation of the topic, can send the data to it and it can finally reach the models for analysis.

4.2.7 Kafka Workflow

Kafka is the control center of the pipeline from where the components send and receive messages. When exchanging messages some components work as publishers and others that work as subscribers. In this pipeline, there are three publish-subscribe relations between components.

(i) **WebRTC Connector → ML models -** streaming data is sent in chunks to the correspondent model for evaluation.

(ii) **ML model → Result aggregator -** prediction results are sent to the result aggregator for later to be stored/sent to the client.

(iii) **Result aggregator → WebRTC connector -** processed results are sent to the connector which already has an established WebSocket connection from where the results are transmitted.

Kafka configurations have to be adapted to the proposed pipeline requirements. To ensure that the system works near real-time the system has to revolve around some characteristics of the data for the different circumstances. Configurations were adopted for Producers, Consumers, and Topics.

Figure 4.8 is a simplified representation that helps to understand and identify the connections described. In this visualization, each link between components is a Kafka publish-subscribe relationship, where the publisher is where the arrow starts and the subscriber is where it ends. On top of the arrows, there is also a name corresponding to the name of the Kafka topic and a number corresponding to the previous enumeration.

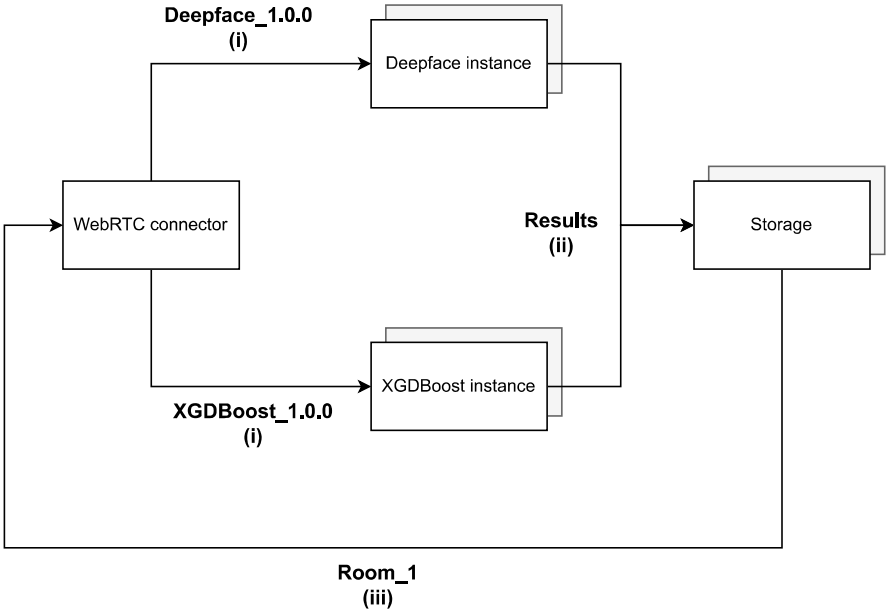

Figure 4.8: Kafka publish-subscribe relations.

WebRTC connector to ML models i

In general, this publish-subscribe relation can be summarized by: A topic for each of the ML models available is created. The WebRTC connector publishes messages with chunks of data from the clients' stream to that topic. ML models poll the messages from their correspondent topic and start the data analysis.

The data that flows in the relation is ephemeral (data is discarded after analysis), disposable (losing messages with some frames of data is tolerable), and unordered (since data is disposable it is not possible to guarantee order without affecting performance).

These characteristics of the data are based on the requirements that the pipeline needs to fulfill. Losing some of the data is not problematic as it does not affect the pipeline or the user's experience and ability to understand the emotion. Ordering data would be ideal for the user to understand the evolution of someone's emotional state, but that could cause problems in terms of parallel processing of messages. So, that results in a scenario where if the user receives a prediction that corresponds to a moment before its current one, it can simply discard it as it is not the latest result.

Based on that the following configurations were selected. For the producers:

- *max.in.flight.requests.per.connection* was set to 1, allowing only one request-in-flight at a time, reducing the chance of message duplication.
- *retries* was set to 0 since data loss is tolerated.
- *acks* was set to 0 which increases the risk of losing messages but decreases latency.

Then, the consumer's main configurations:

- *enable.auto.commit* was disabled for a higher control which ensures that messages are processed only once.

- *max.poll.records* was set to 3 so that the model can queue up to 3 messages at the same time. This can decrease latency.

As for the topic configurations:

- *replication.factor* set to 1 to reduce the overhead of replicating data across brokers.
- min.insync.replicas value to 1 to reduce the minimum required replicas for acknowledging writes. This reduces latency at the cost of potential data loss.

ML model to Result aggregator ii

The machine learning models perform the classification of the input data and then publish the results to the results topic. The result aggregator component receives the messages and stores/processes the results.

In this scenario, data is less flexible. Ordering is not possible as data is already unordered from the previous step and ML models might analyze different chunks in parallel. However, losing data is less tolerable since resources were already dedicated to classifying that data, it is essential to preserve most of the results.

The producer's main configurations consist:

- *acks* set to 1 meaning that the producer does not wait for the acknowledgments from all replicas. This ensures minimal latency while giving a higher message delivery guarantee.
- *linger.ms* set to 0. This forces the producer to send messages without any delay.
- *compression.type* to none reduces latency.

The configurations for the consumer are as follows:

- *fetch.min.bytes* can be set to 1 to decrease latency.
- *enable.auto.commit* was disabled.

Finally, for the topic configurations:

- *replication.factor* was set to 2 for a higher degree of fault tolerance.

Result aggregator to WebRTC connector iii

In this step, data is sent from the results aggregator to the WebRTC connector through a topic dedicated to the specific client. This way, the connector uses the already established WebSocket connection with the client to send the results of the predictions.

As for the Kafka configurations adopted, they are similar to the ones used in the previous publish-subscribe relation since the characteristics of the data and the requirements are identical.

4.2.8 Results storage

The Kafka topic *results* redirects the results from the model serving components to the storage components. This component is not complete as it was not the objective of this work, but the way it is built allows for multiple features to be implemented based on the needs of the pipeline.

Again, the use of Kafka allows the multiple instances of these components to be running at the same time without needing to pay special attention to data duplication. It is also possible to extend the system by creating another consumer to receive the results. As long as this

consumer is part of a different consumer group, it will be able to receive all the results and possibly perform different tasks than the one that is already implemented.

After calculating the output that results from the input values in the message, the model stores in a message the predicted value and the prediction time which is then sent to Kafka again. Storing the information about predictions is essential for further model improvements. Potentially, the storage of this information could be done in the model prediction component, but sending it to Kafka for another component to handle can bring a few advantages. Firstly, if any processing needs to be done in the data that is going to be stored, it does not have any impact on the model prediction component and sets it free to handle more requests. This separation of concerns also provides more flexibility in the pipeline as it allows, for example, different models to have their results stored in different ways without changing any of the code from the models themselves.

Results are returned to the client using the pre-existing WebSocket connection that was established between the WebRTC connector and the browser. There is a topic in Kafka for each of the rooms. The WebRTC connector subscribes to that topic when the client connects and forwards the results to the participant as they appear. Each participant also receives emotions from the other participants in that room. On the webpage, there are two symbols, a microphone, and a camera, that correspond to the audio and video emotions detected for each of the participants.

Figure 4.9 presents an example of how the participants see the results. In this example, these are two fake users from labeled videos.

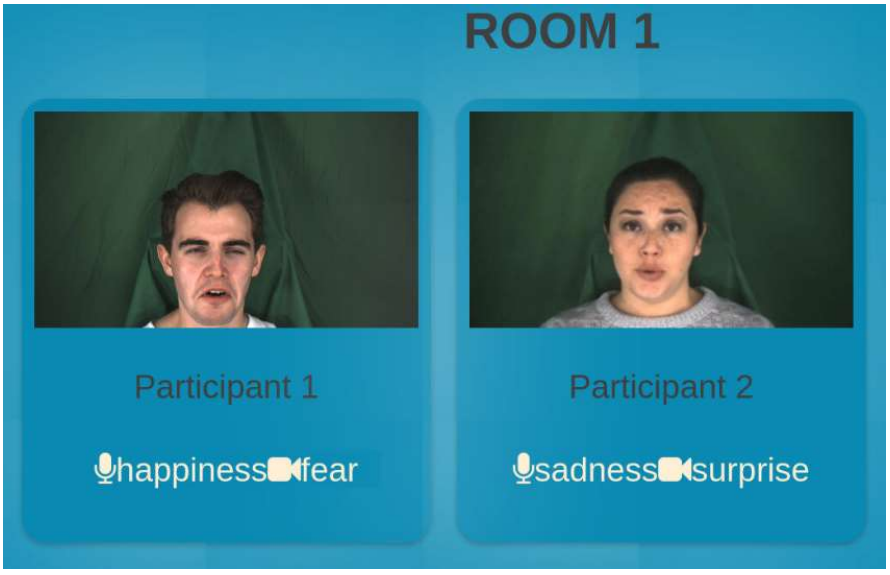

Figure 4.9: Result output in the webpage example.

4.2.9 Monitoring

Ensuring that the pipeline stays functional and finding possible errors is a top priority when designing a system like this. Metric collection on the different components of the pipeline

was done using Prometheus.

Through the Prometheus configuration file, it is possible to specify the addresses where Prometheus needs to collect the metrics. These metrics can be exported in many different ways and depend on the deployment type that is being used. In the local and Kubernetes deployment sections it is explained the metric collection process for each one of them.

Collecting metrics in itself is not very beneficial if they are not easy to understand and draw conclusions from. For that reason, it is important to have concise and meaningful visualizations. Grafana is a tool that has easy integration with Prometheus and allows the creation of dynamic dashboards that improve the visualization and understanding of metrics.

4.3 Local Deployment

While developing the pipeline, a convenient way of testing the system is by having a local deployment based on containers. To speed up the testing process, a docker-compose was used to instantiate the local deployment. This way all the containers needed for the system to run are created with the necessary configurations for them to communicate with each other.

Initially, a Dockerfile was created for each of the components. All the mutable configurations such as addresses were provided using environment variables. Since many of the containers needed to communicate with each other, a docker network was used to work as a bridge between them. The services connect to it and expose only the necessary ports to be accessed outside.

Most services will work with just these steps but others require more configurations.

4.3.1 WebRTC components

The nature of WebRTC connections involves a great number of ports. Containers are running inside an environment that provides a certain degree of isolation and need ports to be exposed to be accessed from the outside. In the case of the media server, it starts a connection with each client that joins a conference. Although docker supports exposing a port range, it uses another approach.

The selected solution is to run the container with network host. This is advised by the Kurento team in their documentation on how to run the Kurento media server in docker. The same problem arises when dealing with the Python aiortc component that connects directly to the client's browser.

However, docker-compose does not allow a service to connect to a network once they are running in network mode host. Due to this situation both of the components run outside the app network and need to connect to other components as if they were in the host machine.

```
kurento:
    image: kurento/kurento-media-server:6.18.0
    container_name: kurento
    network_mode: "host"
```

Code 2: Kurento running in network mode.

4.3.2 Kafka Deployment

Most Kafka deployments rely on Apache Zookeeper[4] to manage all the metadata information about producers, brokers, and consumers. For that reason, it becomes an indispensable service when deploying Kafka. There are already some pre-built images for deploying Kafka that facilitate the deployment and configuration processes. One of the most known ones is the Confluent Kafka.

There can exist more than one Kafka broker running at the same time. Since this is a local deployment and the resources in a single machine can be limited, only a single broker was deployed. Nevertheless, with Zookeeper running, adding more brokers should not be a concern.

For the Kafka docker image, it was also used the one provided by Confluent. This broker needs to register in Zookeeper which is already handled by the image if the Zookeeper address is provided.

This pipeline works on two networks: the one created for the services to communicate and the network host. For that reason, Kafka needs to expose ports to the host and ports to the app network. Addresses and ports need to enable the broker to be accessed from the host and the network app.

The configurations used are presented in the following snippet of code 3.

```
broker:
    image: confluentinc/cp-kafka:7.3.0
    container_name: broker
    networks:
      - app
    ports:
      - "29092:29092"
    expose:
      - "9092"
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
```

Code 3: Kafka broker deployment on docker-compose.

Since most components require Kafka to be running when they start, it was created a health check in the docker-compose to know when Kafka is ready. The components that need to wait for Kafka will wait on that condition and will only start initializing after the condition is met. In this case, the health check is just a netcat command that tries to connect to Kafka and exits when it is successful4.

[4]Apache Zookeeper: `https://zookeeper.apache.org/`

```
    healthcheck:
      test: nc -z broker 29092 || exit -1
      start_period: 5s
      interval: 5s
      timeout: 10s
      retries: 10
```

Code 4: Kafka docker-compose health check.

Another important step in the deployment is the initialization of the Kafka topics. In this system, there are a few topics that need to be created on initialization for every component to work properly. A common strategy is to deploy another Kafka container that waits for Kafka to be ready and executes the necessary commands to create the topics. Afterward, this component goes down and is not needed anymore. The following code snippet shows how this was achieved in this system 5.

```
init-kafka:
  image: confluentinc/cp-kafka:7.3.0
  networks:
    - app
  depends_on:
    broker:
      condition: service_healthy
  entrypoint: [ '/bin/sh', '-c' ]
  command: |
    "
    # blocks until kafka is reachable
    kafka-topics --bootstrap-server broker:9092 --list

    # topics are deleted for debugging purposes
    kafka-topics --bootstrap-server broker:9092 --delete --topic '.*'

    echo -e 'Creating kafka topics'

    kafka-topics --bootstrap-server broker:9092 \
      --create --if-not-exists --topic client --replication-factor 1 --partitions 10
    kafka-topics --bootstrap-server broker:9092 \
      --create --if-not-exists --topic results --replication-factor 1 --partitions 10

    echo -e 'Successfully created the following topics:'
    kafka-topics --bootstrap-server broker:9092 --list
    "
```

Code 5: Kafka-init component.

Topics are being deleted on startup to facilitate the tests.

4.3.3 Monitoring

This docker composes also contains the deployment of Prometheus and Cadvisor for monitoring purposes. Prometheus relies on its configuration file to locate and collect metrics from the running components. Some components export their metrics for Prometheus to collect. This requires that Prometheus knows where to get these metrics. So this configuration file must be correct. This snippet shows the used configuration in Prometheus 6.

```
scrape_configs:
    - job_name: storage
      scrape_interval: 5s
      static_configs:
      - targets:
        - storage:9090
    - job_name: cadvisor
      scrape_interval: 5s
      static_configs:
      - targets:
        - cadvisor:8080
    - job_name: config
      scrape_interval: 5s
      static_configs:
      - targets:
        - config_api:9090
        - config_api:8001
    - job_name: connector
      scrape_interval: 5s
      static_configs:
      - targets:
        - connector:9090
```

Code 6: Kafka docker-compose health check.

An example of the visualizations provided by the cadvisor is pictured in figure 4.10.

Figure 4.10: Visualizations provided by Cadvisor about the metrics of a container.

4.4 Cluster deployment

Local deployment was used to prove that the system would work and meet our requirements. With this solid foundation in place, the system still has yet to reach a state where it can scale

and efficiently manage the resources to accommodate increasing user demands. With the adoption of Kubernetes, the system can take full advantage of its automatic load balancing, horizontal scaling, and fault tolerance capabilities.

The cluster deployment was tested locally using Minikube[5], a simple Kubernetes implementation that makes it possible to develop for Kubernetes locally. Although Minikube is not suitable for production environments, it was not possible to test the deployment in a production cluster.

Since Kubernetes uses and manages containers, the images created in the local deployment were adapted without suffering many alterations.

Figure 4.11 provides a simplified overview of the cluster deployment. The orange links represent the export of metrics from the various components. Strimzi represents multiple components that are required for deploying Kafka.

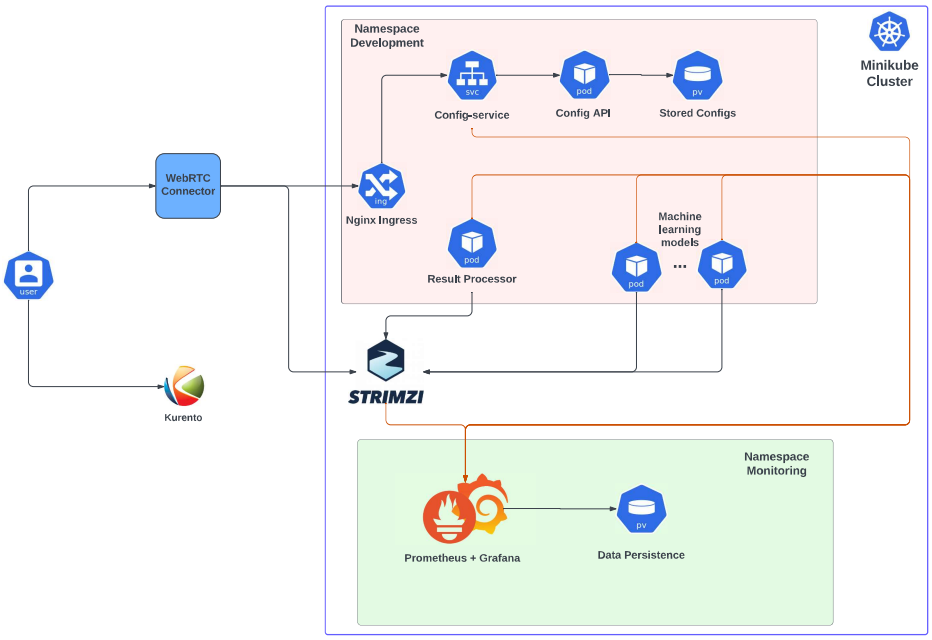

Figure 4.11: Cluster deployment diagram.

4.4.1 WebRTC in Kubernetes

During the research and deployment of the WebRTC components in Kubernetes, we reached the realization that WebRTC applications are not meant to be deployed inside Kubernetes, at least not in the usual way Kubernetes handles the rest of the deployments. The problematic aspect of WebRTC is that it needs a big port range to be continuously open. Kubernetes in a normal scenario uses an ingress to serve as an entry point to the cluster which only allows access through some defined ports.

[5]Minikube: `https://minikube.sigs.k8s.io/docs/`

There are solutions but none of them are standardized or optimal. This was an aspect that was already taken into consideration when the decision to adopt Kubernetes was made.

Port range

WebRTC requires a huge port range to function properly. Opening a high number of ports in the cluster would allow clients to communicate from the outside. This would, however, be detrimental in terms of performance as Kubernetes is not built for that purpose and could be problematic for security reasons. There exists evidence that this approach works but the implications and problems that can arise from it make it not desirable.

STUNner

Due to the lack of solutions and business opportunities that it creates, an open-source project emerged that tried to solve the issue. STUNner's objectives are focused on integrating WebRTC media servers with Kubernetes by providing a deployment that has one access point to the cluster but that allows many clients to connect to it simultaneously.

Since this project seemed promising, some work was done to integrate it into our system. In figure 4.12, taken from STUNner documentation, it is visible that there is an entry point component that serves as a proxy between the client and the media server.

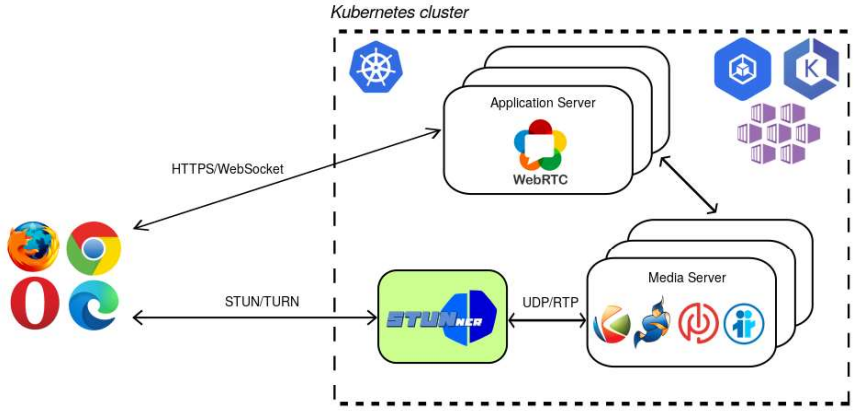

Figure 4.12: STUNner architecture taken from the online documentation [6]

Due to that, the client needs the address of the entry point instead of the media server one. STUNner has a library built in Javascript that handles those changes for us. However, our implementation of Kurento is built with Java.

To test whether STUNner would work, it was built with the same Kurento application but this time using Javascript. Unfortunately, the Kurento team did not create the group call example in Javascript, so it needed to be built from the beginning.

Afterward, using their library and deploying the needed components in Kubernetes following the tutorials available in their documentation, it was not possible to establish a connection. This solution was then discarded considering that the product is new and thus

[6]STUNner github: `https://github.com/l7mp/stunner`

there is not much support for it. Nevertheless, in the future, this solution seems very promising and would potentially solve this issue.

Outside deployment

In the end, the presented solution deviates from the initial ideas. Since the options that were found did not satisfy the goals or were unfeasible in the available time, the solution was simplified.

The deployment and scalability of the WebRTC components in Kubernetes were moved to future work. To test the pipeline some changes were made to the deployments. First, all the WebRTC components were deployed outside the cluster. Both Kafka and the configuration API were made accessible outside the cluster so that those components could connect to it. Strimzi Kafka operator can easily do this by changing a simple configuration. In the case of the configuration API, it was introduced an Ingress in the cluster to forward the traffic to the configuration API service which in turn serves the multiple instances.

The WebRTC components are executed outside the cluster and their configurations are adapted so that they can access the cluster components. This has some drawbacks and limitations, noticeably the lack of scalability in both the WebRTC connector and Kurento. In the future, there would be necessary the introduction components capable of managing the multiple instances of the WebRTC components.

4.4.2 Components' deployment

Before deploying the components, it is important to store the images in an image registry so that the cluster can access them. The system's images were stored in the GitHub registry since the codebase is already in GitHub. The project is private so it was necessary to create a secret in Kubernetes for it to be able to pull the images from the registry.

Each component has its own Kubernetes deployment and since they are all stateless, it is possible to define the number of replicas. There were a few components that needed extra configurations.

Configuration API

In Kubernetes, the best way to access a deployment is through a service. The configuration API component is accessed from the other components so it makes sense to put a service in front of it. The service also provides load balancing capabilities which allows the component to be scaled.

As previously mentioned, the WebRTC components outside the cluster need to connect to this component. Using an ingress, all the traffic that targeted the configuration API was redirected to it.

Kafka

Kafka deployment is different from the others. Although Kafka deployments can be created, it is much easier to use operators that already do most of the work for us. It used the Strimzi Kafka operator to deploy Kafka in the cluster.

Strimzi deployments are highly configurable and need to be adjusted to the needs of the system. The WebRTC components need to connect to Kafka hence the configurations in the operator deployment were changed so that those components can access Kafka from the outside. By changing that configuration, Kafka exposes the brokers externally and it is given a fixed address for other components to connect to. As the components inside the cluster also need to connect to Kafka, it makes sense to make it accessible for them as well. Those were the most important configurations that were altered from the base Strimzi deployment.

4.4.3 Scalability

Differently from the local deployment, using a Kubernetes cluster creates the opportunity to auto-scale the deployed components. The main components that need scaling are the model-serving ones. As machine learning models can be slow in comparison to the other components in the system, they are most likely going to be the bottleneck of the system. Adjusting the number of running instances of a specific model according to the load is essential to keep the response times as low as possible.

However, we found a problem related to the auto-scaling of these models. Models receive their input as messages from their own Kafka topic. If there are multiple consumers in that same topic(these consumers are in the same consumer group), Kafka will assign the partitions to those instances. But assuming that the load can increase drastically and the number of running instances spikes, it can surpass the number of partitions in that topic. This creates a scenario where there are instances that will not have a partition assigned to it, they will become idle. The number of partitions is set whenever the topic is created but it can be updated afterwards. If the number of partitions is set to be higher than the expected demand, there will be no problem. For the cases where it is not possible to do so, the number of partitions needs to be updated to match the number of instances.

A new component was developed to solve these edge cases. The component is responsible for monitoring the number of running instances and updating the number of partitions in a topic. Since the models are deployed in Kubernetes, this new component uses the Kubernetes API to check how many instances are running. If the number of partitions is not sufficient, it increases it. Important to note that there a some aspects of updating partitions that need to be taken into consideration. First, the process of updating it is expensive and might impact the performance of Kafka. To minimize this impact, when partitions are updated, it is added several partitions that overestimate the expected number of instances. This way, they are updated less frequently. Secondly, Kafka does not support decreasing the number of partitions. Having more than the desired partitions can have implications in terms of performance and resource usage, but it is not as problematic.

Due to the referred limitations and that it would introduce more complexity to the system, this approach was dropped.

The opted solution discards the possibility of scaling indefinitely. It starts the topics with a partition count based on the model. If the model is expected to have a high number of instances, then on creation the topic should already take into account that demand, and it

should be initialized with a matching number of partitions. This greatly simplifies the problem and does not affect many use cases. Overall, although solutions can be integrated, they do not remove the problems from one another and would need extra effort to make it better.

To scale the model serving components, it was used the Kubernetes autoscaler. Kubernetes collects metrics from the running deployments. By specifying a metric and the threshold value for it, Kubernetes will instantiate more replicas when that threshold is reached. It also supports scaling down whenever it sees that some instances are not needed. In our scenario, new instances are created based on the CPU usage. By going over the CPU threshold(which varies depending on the model), the autoscaler creates another instance. If the CPU usage decreases, the instances will be removed after 10 minutes.

4.4.4 Monitoring

The monitoring environment in Kubernetes is much more mature in comparison with local deployments. It already has many monitoring features built in. Similarly to the local deployment, Prometheus is responsible for collecting and storing the metrics. The visualization of metrics is enhanced by Grafana by providing easy ways of building charts and other visualizations.

Prometheus Setup

Initially, Prometheus was deployed in the cluster and made available to the other components. Since volumes and persistence of data in general work differently in Kubernetes, a StorageClass is required to persist the data collected by Prometheus. Rancher[7] was used as a storage provider. It enables Prometheus to have access to persistent storage through a PersistentVolumeClaim.

The configurations are available to Prometheus as a ConfigMap. These configurations have to be adapted to the Kubernetes deployment. A service for the Storage and Config API exposes their metrics for Prometheus to collect. This allows the collection of metrics from many pods of the same component without requiring any changes to the configurations.

For Kafka, metrics can be exported by changing the configurations in the Strimzi operator. It provides metrics such as the number of topics available, the number of partitions, and resource usage by each Kafka node.

The common metrics such as CPU and memory usage for each component are made available using KubeState metrics, an addon that listens to the Kubernetes API server and generates metrics.

Grafana Setup

Grafana is deployed with its own PersistentVolumeClaim and Service. Although it can get data from multiple sources, in our case it is only using Prometheus as a data source.

For some components, it was used pre-existing dashboards that are available for public use. For Kafka, Strimzi offers a Grafana dashboard that contains representations of the metrics

[7]Rancher: `https://www.rancher.com/`

that it exposes such as the number of topics, partitions, and consumer groups. For KubeState metrics, it also exists some built dashboards for visualizing Kubernetes data such as GPU and memory usage, and the number of pods.

Since some of the metrics in the pipeline are exported by the components that were implemented, another dashboard was created to display those metrics. It contains information about the predictions by model type, model execution times, number of processed requests, etc.

CHAPTER 5

System Validation

To conclude whether the system fulfills its requirements, multiple tests were made to replicate scenarios the system should satisfy. By monitoring the individual components and analyzing the system's output, it is possible to conclude whether the solution met the expectations.

The pipeline was exposed to each use case, and the results were collected to compare it to the expected outcome.

Since our solution runs locally for both the docker-compose deployment and the cluster deployment, there will not be a distinction for these scenarios aside from the use cases specifically targeting a functionality only available in one of them, for example, the auto-scaling capabilities.

5.1 System testing setup

The testing setup must be explained before describing the tests and presenting their results. There was a focus on preserving the same characteristics from test to test to ensure their reliability. Many variables in the pipeline can influence its output. This setup tries to minimize the influence of these variables by having similar scenarios in each test case. Since the pipeline input comes from the user's camera and microphone, comparing use cases with different inputs can be challenging. It is then essential to replicate the same inputs to obtain comparable results.

We used browser drivers to simulate a participant using the video conference system. Using Selenium[1], WebDriverIO [2], and Chrome/Firefox web drivers, the testing setup instantiates many browser clients that connect to the pipeline via the web page.

Clients are running instances of the web browsers and interact with the page by executing the instructions provided in the code. They simulate a participant joining a room and connecting their camera and microphone. Selenium manages those clients and provides an

[1]Selenium: `https://www.selenium.dev/`
[2]WebDriverIO: `https://webdriver.io/`

interface to see and interact with each client's webpage. A docker-compose is responsible for instantiating the clients and Selenium. The number of clients can be carefully selected using the docker-compose scaling functionalities for each use case.

In regards to the used web drivers, initially, the testing framework used the Firefox web driver to mimic a participant's behavior. With the correct configurations, the web driver can fake the camera and microphone, replacing them with a random noise video and no audio. It could be used for a load testing use case but proved useless in other scenarios.

Chrome web driver was more beneficial as it can use pre-recorded videos as fake input for the webcam and microphone. Providing a file that is already labeled not only gives better replicability but also enables conclusions to be taken about the output of the machine learning models used.

Although the testing framework is automatic, manual tests, where someone connects to the pipeline via the webpage, can still be executed simultaneously.

Figure 5.1 describes how the setup is made. Only the entrances of the system are depicted here, however, the rest works the same way it does for a real participant.

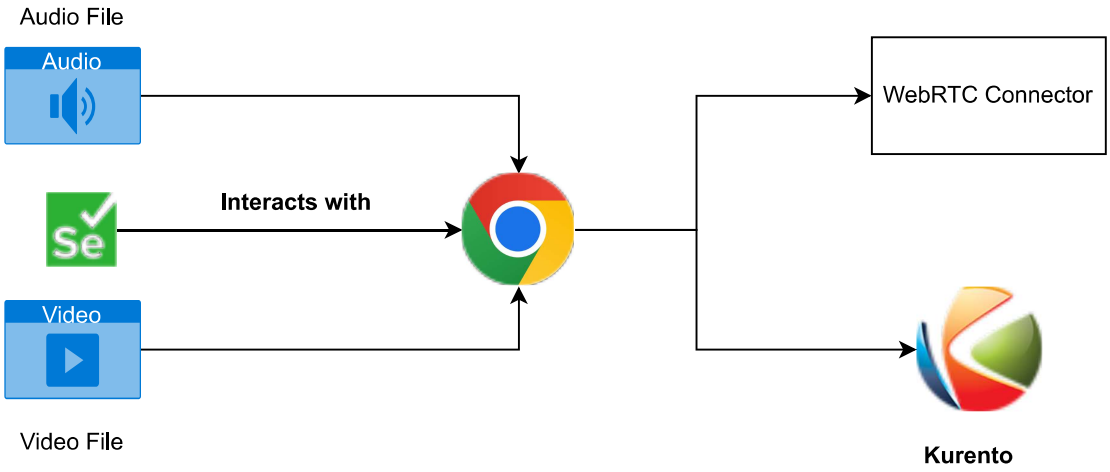

Figure 5.1: Test set up to create a fake video conference participant.

Figure 5.2 shows the interface provided by Selenium where there are three active sessions. Each session represents a participant and by clicking on one of them, the correspondent web interface can be seen.

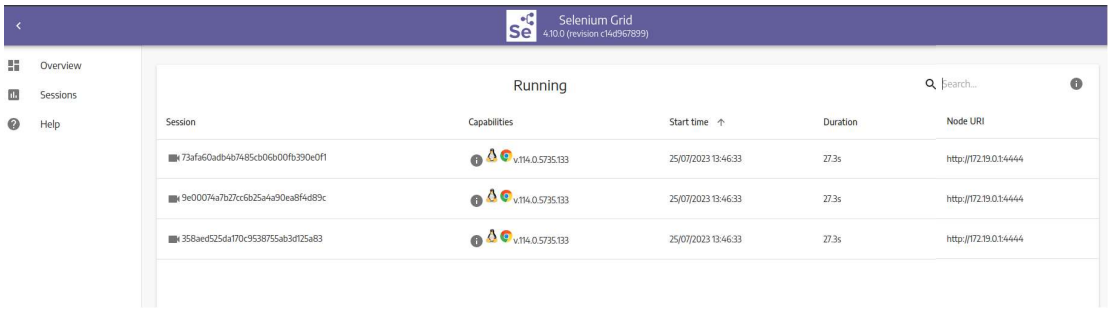

Figure 5.2: Selenium interface showing three client sessions running.

Our system uses different machine learning classifiers that try to understand the user's emotions. Some receive audio as input while others analyze video frames. The development of these models was not part of this work and the supplied models were used without any modification. This section briefly describes the models used for testing the pipeline.

5.2.1 Deepface

Deepface is a framework, available as a Python library, that includes state-of-the-art machine learning models for face recognition[56]. Moreover, it contains more functionalities such as age, gender, race, and more importantly emotion recognition[57]. The classifier acts upon images and upon detecting a face, outputs the probability for each emotion. The model supports six emotions: Angry, Disgust, Fear, Happy, Sad, Surprise, and Neutral. It is a model based on Deep CNN and was trained with the FER-2013[3] dataset. Some preprocessing can be made to the images to obtain better features and thus obtain better results from the model. It starts by detecting and cropping the faces of the image. Then it is possible to use face alignment where many transformations are made to the face to make it more suitable for analysis. Face alignment is not a required step and can be skipped. Lastly, the face is analyzed and the respective emotion is returned by a machine learning model.

5.2.2 CNN based Image classifier

The research team developed a machine learning model based on a CNN to be experimented on. The face recognition pipeline is a simplified version of the Deepface one, skipping some of the steps. There is no alignment or normalization of the input.

Using the OpenCV library, images are cropped to contain only the outlined face that was detected. Note that the model only makes predictions on one of the faces in the image, discarding the others. The cropped image containing the detected face is fed into the model for classification. The output of the model is a vector corresponding to seven emotions: angry, disgusted, fear, happy, sad, surprised, neutral.

Three variations of the model were created using different input sizes: 48x48, 60x60, and 100x100. The MEAD[58] dataset was used for training.

5.2.3 XGBoost Audio Classifier

For the audio classifiers, will be used an in house classifier. The developed classifier uses eXtreme Gradient Boosting (XGBoost) which uses custom-selected features retrieved from an audio segment to predict the user's emotion. Features were carefully selected, resulting in a classifier that tries to maximize accuracy with a small number of features which decreases the prediction time. The classifier was trained using the Interactive Emotional Dyadic Emotion Capture (IEMOCAP)[4] database. It is a collection of audiovisual data with annotated emotions. Actors total ten that are evenly distributed by sex and speak fluent English.

[3]FER-2013: `https://www.kaggle.com/datasets/msambare/fer2013`
[4]IEMOCAP: `https://sail.usc.edu/iemocap/`

5.2.4 Neural Network based Audio Classifier

We have also selected an open-source classifier[59][5]. This classifier is based on Deep neural network and uses Mel-Frequency Cepstral Coefficients(MFCC) as input instead of the handcrafted features.

The results will not be focused on this model but we integrated an open-source model to test the extensibility of the pipeline. The integration processed ended up not being much different from the other models already described.

5.3 Results

Initially, the first tests seek to gather results about the overall delay of the system. The expectations were that the results would vary based on the classifier and its configurations. For that reason, the pipeline was subjected to independent tests for each of the available models. The configurations used for the classifiers will have an impact on the results obtained but for the moment it was selected only a set of configurations for each. The reproducibility of the tests was assured by the common testing setup.

The tests were all performed on the same machine with the following specs.

- CPU - 12th Gen Intel(R) Core(TM) i7-1255U
- GPU - Iris Xe Graphics 46a8
- RAM - 32GB

5.3.1 Pipeline performance

Upon deploying the pipeline in Kubernetes, the first measurements observed were the memory and CPU resources that it requires to run. This measurements were obtained for two different scenarios.

The first is where the pipeline only has the fixed components, which correspond to the components that cannot be changed. Essentially, everything is running except the classifiers. Taken from one of the Grafana charts, figure 5.3 shows the CPU and memory usage of the pipeline when idle.

The results show that the system uses 5.36GB of memory, 5 out of 12 CPU cores, with a total of 28 Kubernetes pods.

[5]NN classifier: `https://github.com/marcogdepinto/emotion-classification-from-audio-files`

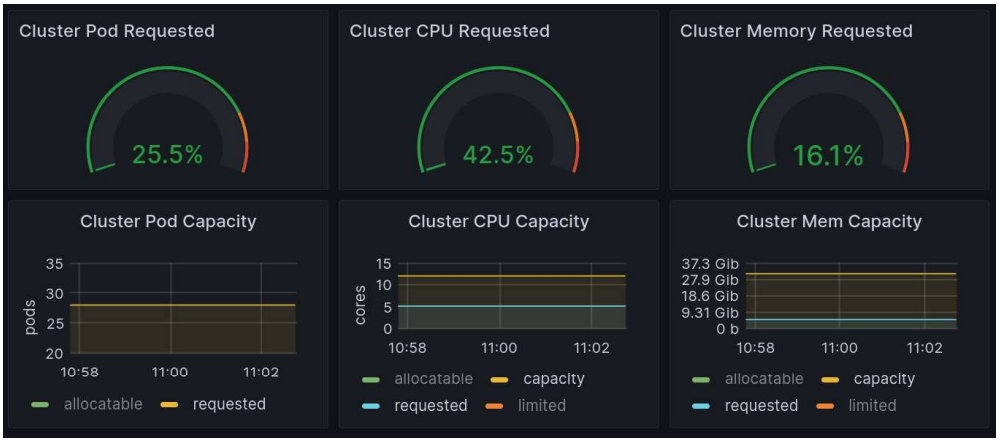

Figure 5.3: Pipeline resource consumption when idle.

5.3.2 Model performance

After running the pipeline and the testing framework with the characteristics of the specified use case, the results were obtained from Prometheus/Grafana and the stored values in the Storage component.

A set of measurements were collected.

- **Model configurations** is specific to the configurations that were provided to each model. It is a fixed value that will only depend on the configuration values in use.
- **Average prediction time** is the time taken for the classifier to obtain a result.
- **Pipeline delay** is a group of delays that are hard to measure individually. It consists of message travel times, time spent waiting for resources(most notably, waiting for a classifier to be free), and other less significant delays added by the pipeline. This value is obtained after measuring the total delay.
- **Total delay** is measured at the end of the pipeline.

Every classifier was subjected to those test case scenarios and the results were also compared to the same classifier running outside the pipeline.

Image classifiers

Both image classifiers were subjected to multiple test cases to compare their performance in different scenarios. The first step was to make a replica for each classifier that runs outside the pipeline. Since this replica is not part of the pipeline and is tested outside that environment, the results of the tests performed on it will serve as a base for comparison.

Both models have characteristics that were taken into consideration in the tests performed. The image emotion classification is not a single task but several sequential steps, of which some can be skipped. The time it takes to perform one of these steps can depend on the size of the image provided. Thus, the first experiments try to understand the influence that the image size has on the performance of the model.

The tests performed on a standalone model, running outside the pipeline, showed a correlation between the model's prediction time and the size of the input image. The sizes

selected were based on the common video resolutions: 480p(640x480), 720p(1280x720), and 1080p(1920x1080).

The results for the average prediction time for each image size are presented in the table 5.1. These times reflect the prediction times obtained for classifying 570 images/frames of the correspondent size.

Table 5.1: Average prediction times(ms) for image classifiers on different image sizes.

	1920x1080	1280x720	640x480
Deepface with Align	78.84	54.02	46.82
Deepface without Align	62.67	47.219	39.42
CNN	39.88	35.24	31.07

As expected, images with higher resolution take more time to process. Another conclusion taken from the results is that the DeepCNN is slightly faster than the Deepface model, even without the Alignment. This is most likely because Deepface contains other steps and verifications that DeepCNN did not implement. After this point, the majority of the results are focused on the Deepface classifier.

An individual script runs sequentially and, differently from the pipeline, is not expected to receive concurrent requests for processing. In the pipeline, the classifier's prediction time is much more valuable. Delays add up and become a bottleneck. However, if the model configurations are properly selected there might be a way of mitigating the problem. For this particular model type, the number of frames that are received and the size of the frames are the most detrimental to the good function of the pipeline.

Starting with the selection of the number of step frames. The pipeline only analyses every n frame(the value of the step). It is easy to understand that if a classifier takes 100 milliseconds to analyze a frame, then it can only receive frames at a rate of 10 fps. Increasing the fps would increase the pipeline delay. Tests were performed at 30 fps. Testing with multiple values granted the results displayed in figures 5.4 5.5.

The measurements obtained are as predicted. By increasing the number of stepped frames, which in turn reduces the number of frames analyzed per second, the total delay decreases5.4. The image size also impacts the pipeline's performance and can influence the selection of the number of stepped frames.

The prediction times are related to the size of the images but seem to be independent of the number of stepped frames5.5. In comparison with the results obtained for the same classifier running inside a script outside the pipeline, the newly obtained results are significantly higher. The main reason might be due to the increased load that the testing machine is exposed to. However, it is important to examine that the results are comparatively the same, the bigger the image size the more time it takes for the model to make predictions.

The values of these parameters have an enormous impact on the performance and usability of the pipeline. Figure 5.6 demonstrates one of the implications of what can happen to the pipeline if these values are not carefully selected. In the example, the number of stepped frames is not enough to reduce the overload on the pipeline, causing many frames to be queued

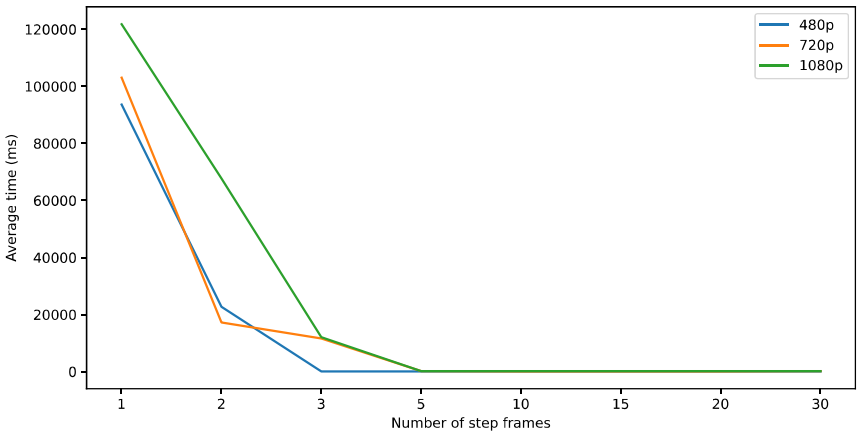

Figure 5.4: Deepface classifier Total time measurements for different step frames for each image resolution.

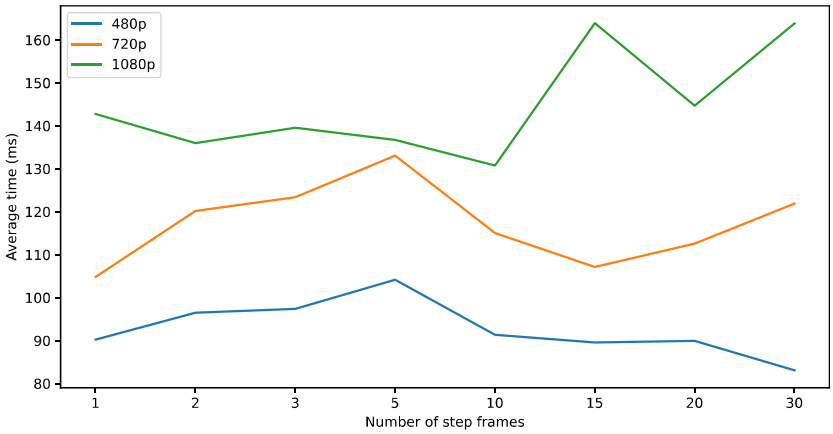

Figure 5.5: Deepface classifier Prediction time measurements for different step frames for each image resolution.

while waiting for a classifier to be available. The delay keeps accumulating and many of the frames will eventually have to be discarded. Figure 5.7 shows a much better relation between the prediction time and the total time in the pipeline. Note that for testing purposes, frames are not being discarded after a certain threshold.

One individual classifier might not be enough to handle all the requests. For that reason, the classifiers can be scaled using Kubernetes to increase the number of concurrent requests that are answered.

A new batch of tests was created to answer the questions:

- What is the impact of adding more users to the performance of the pipeline?
- What is the impact of adding more instances to the performance of the pipeline?
- How does the auto-scaler handle the load?

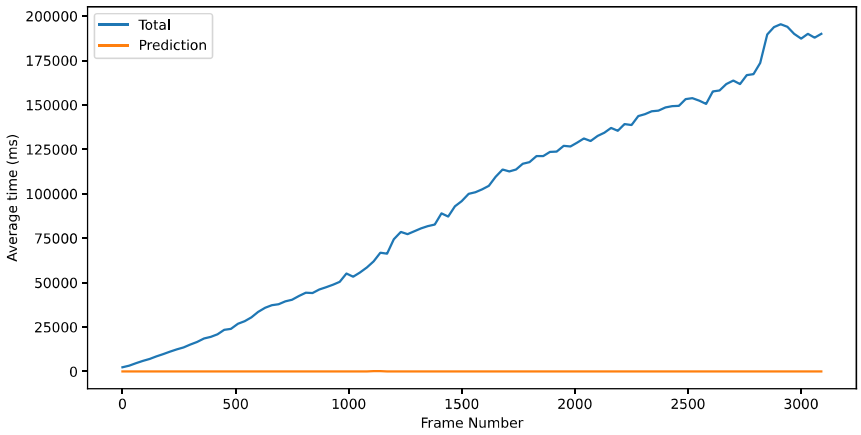

Figure 5.6: Deepface classifier time measurements for 480p resolution with step of 1.

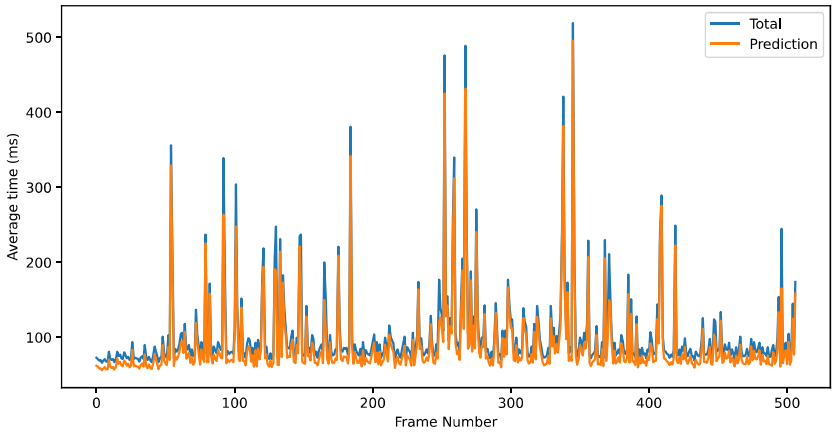

Figure 5.7: Deepface classifier time measurements for 480p resolution with step of 15.

The pipeline was executed with specific configurations that were picked to better visualize the results. For the first question, the Deepface model was set to have a step of 10, and the images provided were in 480p resolution. This choice was based on the observations made previously that are presented in figure 5.6. With these specific configurations, it was observed that the pipeline could handle one client without inducing an incremental delay. The test used the same scenario but with the addition of two new participants, totaling three participants running at the same time. In table 5.2 are presented the measurements of both test runs.

With the information provided, it is possible to infer that the addition of more clients does impact significantly the performance of the pipeline. The average total increase represents the average amount of milliseconds added per frame. A positive number means that each new frame increments the delay observed in the last measured frame. In essence, unless there is some method to ignore old frames, the total delay will approach infinity. The pipeline can handle one client without adding delay for each new frame. However, if the clients scale up

Table 5.2: Pipeline measured times (ms) using three clients and one deepface model instance.

	1 client	3 client
Avg Total	288.91	50426.13
Avg Total Increase	0.89	124.19
Avg Prediction	89.11	160.96
Avg Prediction Increase	0.07	0.19

to three there is a noticeable delay that is added at each new frame. Since the classifier is not able to respond to the high amount requests, many of them will be stuck waiting and thus will take more time to predict. The prediction stays constant on both scenarios but the increase load in the machine leads to a higher average prediction time with more clients. The tests were performed with a given set of configurations, but in theory, the same would apply to any other set of parameters.

The second test scenario measures the impact of the model scaler. Looking at past results, this time the tests were performed with a new set of configuration parameters. The model step was set to 1, meaning that no frames are being skipped, and one client. The results compare two test runs, one where there is only one instance of the model and another where the model was scaled to have three active instances. Measurements obtained are displayed in table 5.3.

Table 5.3: Pipeline measured times (ms) using one client and three deepface model instances.

	1 instance	3 instances
Avg Total	25079.28	5057.91
Avg Total Increase	53.73	10.85
Avg Prediction	80.34	123.30
Avg Prediction Increase	0.01	0.04

These tests try to understand the scenario where the model prediction is slower than the arrival of newer input. With only one instance, the pipeline with these specific parameters was not able to handle the requests in time 5.6. For that reason, the total pipeline delay kept increasing as more messages were arriving.

By adding more model instances, the pipeline distributed the workload through the available instances, that are now able to predict concurrently, decreasing the overall delay of the pipeline. However, it was recorded an increase in the average prediction time. A plausible explanation for this effect can be attributed to the attached extra processing power required to run more instances.

Lastly, it was tested the auto-scaler ability to scale the classifiers to handle the extra demand. Due to the limited computational power available, the tests were simplified. Initially, one classifier was made available. A set of 5 clients was initialized to induce load on the pipeline, expecting the auto-scaler to create more instances. Figures 5.8 and 5.9 show the pipeline pods running before and after the clients were initialized.

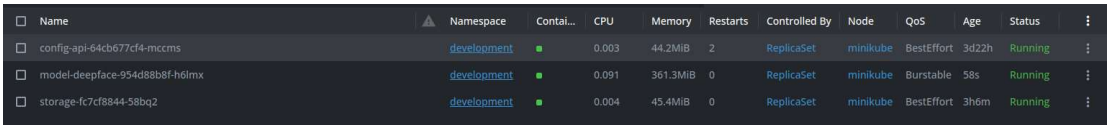

Figure 5.8: Deepface auto-scaler test before the clients have been created.

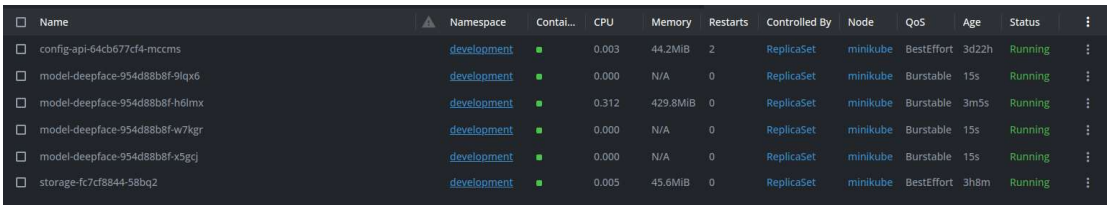

Figure 5.9: Deepface auto-scaler test after the clients have been created.

Audio Classifiers

Both image classifiers make predictions on a single frame, so skipping frames is a valid approach for improving the performance. On the contrary, audio classifiers need to receive a continuous stream of data. The configurations provide the developer the ability to control the size of the chunk of data that is sent to prediction.

Initially, a similar approach was used for testing the audio classifiers. A script was executed outside the pipeline to measure their execution times. The tests ran for 500 times for the same input file, trimmed to be of a specific chunk size. Values can be observed in table 5.4.

Table 5.4: XGBoost classifier measured prediction times outside the pipeline for different input size chunks.

	Chunk 1s	Chunk 2s	Chunk 3s	Chunk 4s
Prediction Time	140.62	156.54	164.86	185.76

There is a variation in the prediction times for different chunk sizes, but in general, the values are close to each other.

Although, in the image classifiers, the selection of a high step number would induce a delay to the client (if frames are skipped then the client stays more time waiting for a response), this delay is considerably small for realistic values of the step parameter. However, audio classifiers have to wait for the complete chunk of data to be ready. Based on the training dataset used for the XGBoost classifier, the average input size for this model is close to three seconds. Note that it is possible to provide a different input size because this classifier does not predict the audio segment itself but on a fixed size amount of features derived from that segment. Thus, it is only possible to change this value for this classifier.

The size of the chunks has a big impact on the client's perception of delay because it is necessary to wait for the input audio that the client provides. This way, the minimum delay that the client waits for is always going to be higher than the chunk size. That is, if the parameter is set to 3 seconds, the pipeline collects data for that amount and only then sends it for prediction. From the client's perspective, the results will have a delay of at least 3 seconds. Most likely this ends up having the biggest impact on the delay. In table 5.5, it was

measured the impact that is added by the pipeline itself.

Table 5.5: Average delay measurements (ms) for XGBoost audio classifier for different chunk input sizes.

	Chunk 1s	Chunk 2s	Chunk 3s	Chunk 4s
Avg. Pred. time	152.60	196.71	238.43	309.31
Total time	163.37 (+10.77)	215.01 (+18.30)	256.46 (+18.03)	324.36 (+15.05)

What can observed is that, contrary to the image classifiers, the prediction time is the biggest factor in terms of added delay. Since input takes much more time to arrive than the prediction itself, the total delay is very similar to the time it takes to predict. In comparison, to the image classifiers, this model takes more to make a prediction which is expected as it analyses a larger input. Also in comparison with the results for the prediction times obtained from outside the pipeline, it is observed an increase in the prediction times for larger input. The main explanation for this can be attributed to the extra load imposed on the machine.

The Kubernetes scaler was also tested. The pipeline was observed when running an auto scaler with one replica that could scale to five replicas. Then, five clients were instantiated. Figures 5.10 and 5.11 are screenshots taken from Lens where is it possible to see the pods that were running before and after, respectively.

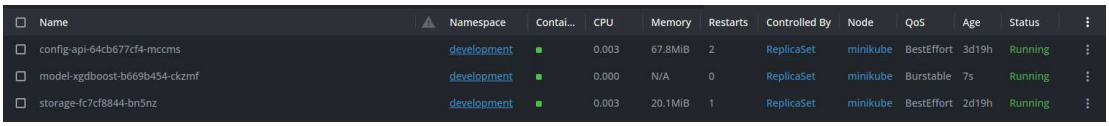

Figure 5.10: XGBoost auto-scaler test before the clients have been created.

Figure 5.11: XGBoost auto-scaler test after the clients have been created.

Since the auto-scaler is based on the CPU metrics collected by Kubernetes, if the pods go over the threshold (in this case it was 90% CPU usage), another instance will be created. However, it does not seem to be a good way of measuring the load. Perhaps an alternative based on the number of messages in Kafka would be better.

5.3.3 Model Evaluation

Although improving the models at use is outside the scope of this work, it is important to evaluate their integration with the rest of the pipeline. The correctness of the output in terms of how accurately it predicts the displayed emotion is not the goal of the tests. A pre-recorded video was used as input to the platform to understand if the system is providing consistent and useful data to the model serving components.

It is expected that the results are similar to the ones obtained by a script running outside the pipeline. Using the testing framework it was possible to select a set of audios and videos that will serve as a comparison between the classifier in the pipeline and the same classifier outside the pipeline.

However, it is important to keep in mind that both audio and video suffer some processing as they go through the pipeline. Things such as changing the audio sampling rate, resizing the image, etc.

For the Deepface classifier, it was used the results gathered for step 1 (every frame was analyzed) and then compared with the results of the script for the same video input. Table 5.6 presents the results. The number of frames analysed was not the same but since the image is running on a loop, it is expected that the results are periodic.

Table 5.6: Model evaluation results for the Deepface classifier.

	Neutral	Fear	Sad
Pipeline results (%)	73.6%	25.2%	1.2%
Script results (%)	67.8%	30.4%	1.7%

Only three emotions were predicted in both scenarios but the percentages of classification were not the same. The percentages are, however, relative to each other, the biggest one being neutral, followed by fear, and then sad representing a small portion of both scenarios.

It is plausible that the extra processing done to the input can be the major factor here since the results are not far apart.

As for the audio classifiers, the results were not as good. There was a noticeable distinction between the results taken from inside and outside the pipeline. We then opted to investigate the audio data that was being analyzed. The pipeline records the audio and saves it into a file. After listening to both the original and the one taken from the pipeline, they seemed to be the same. Then, we converted the audio to a spectrogram and observed that there were some subtle changes in the audio. Figure 5.12 displays the spectrograms of both audios side by side. It is possible to observe that they are very similar but there are some parts in the pipeline audio where the data is smaller. This might be due to conversions in the sampling rate that occur in the pipeline. Although a human can hear the audio in the same way, the classifier is very susceptible to these variations, and thus, this is to be further investigated.

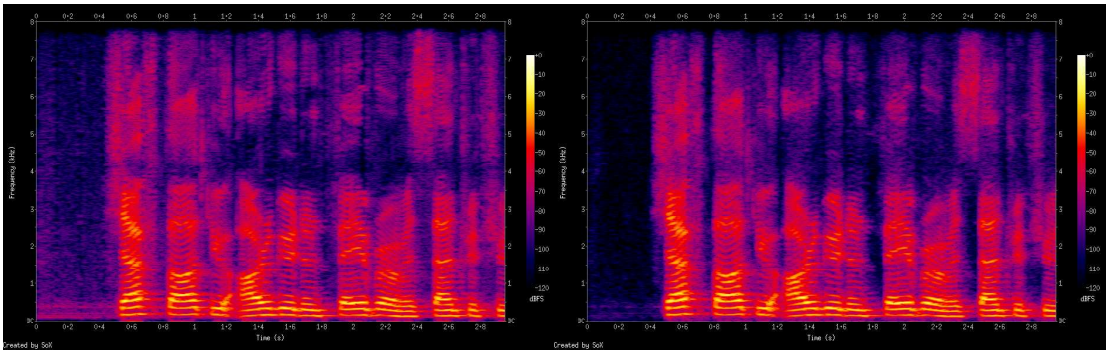

Figure 5.12: Comparison of the spectrograms from original audio(left) and the audio taken from the pipeline(right)

5.4 Limitations

There were many limitations when it came to testing the pipeline and its components. Firstly, the tests were executed on a single machine, which although powerful, cannot be compared to a cluster with many nodes. For that reason, some of the results might be a bit shifted from reality, such as the prediction times when there is more load in the machine. The use of a single machine leads to the incapability of exploring more results. It was not possible to test the pipeline for a high number of users and thus the results were merely speculative.

In WebRTC it is possible to specify some of the parameters for the input video streams. Those can be video resolution and frame rate. However, WebRTC in a real-world scenario, will adjust the parameters according to the machine's capabilities and internet connection speed. In that scenario, the results would be harder to predict. The solution for maintaining the resolution was to enforce them on the code by reshaping the image size for the resolution at test. This adds delay but keeps the results comparable between the test runs. As for the frame rate, it was specified in the WebRTC connection but there isn't a guarantee that this parameter did not change.

CHAPTER 6

Conclusion

6.1 Implemented solution

This dissertation explored the development of a proof of concept extension to a video conference system that includes near real-time feedback of the participants' emotions, using video and audio. The system makes use of existing machine learning models that are capable of analyzing and classifying emotions based on audio and video features.

A system architecture was conceived to connect a video conference platform with machine learning models. Participants use the video conference platform as usual and the proposed system would extend the media server to provide sentiment analysis on the video conferencing sessions. Behind the scenes, the system offers an easy way of integrating new machine learning models, by supporting configurations that change the behavior based on the requirements of those models.

The main challenge of such a system was the necessity of ensuring fast and reliable ways to classify segments of audio and video data in near real-time. Decreasing the delay within the system was one of the hardest obstacles to overcome. The architecture was well tough to overcome the challenges while still providing the flexibility and extensibility required.

The proposed proof of concept solution incorporated a video conference platform that would handle interaction between participants. A component was created to receive and aggregate in chunks the audio and video streams from the participant through a WebRTC connection. Kafka would distribute the chunks across the machine learning instances for prediction. Classification results are stored in a separate component and sent to the participant.

The experiments focused on evaluating the performance and correctness of the system. Many model configurations were selected to observe the influence that they had on the system. Load tests measured the system's ability to scale by adding more instances to accommodate that load. The correctness of the input provided by the system was evaluated by comparing the results of a model running outside and inside the system environment. Those results showed that slight modifications are happening to the data, but small enough that the results stay similar.

6.2 Future Work

During the development of this work, there were aspects that we were not able to fulfill in the available time. Although the system behaves according to requirements there are still some improvements that were identified that can fix some of the problems.

The addition of a service discovery component that works together with the configuration API already developed could provide a better way of managing the model instances created and potentially could help find a better metric to scale the machine learning models.

In the implementation, it was mentioned that the Storage component was just used as a proof of concept and there could be an effort in extending and improving that to use a time series database. In terms of model configurations, many more could be explored and the existing ones have room for improvement.

The next big step falls on the deployment of the WebRTC components in a scalable environment.

Referências

[1] Y. Khaireddin and Z. Chen, *Facial emotion recognition: State of the art performance on fer2013*, May 2021. [Online]. Available: https://arxiv.org/abs/2105.03588.

[2] X. Cai, J. Yuan, R. Zheng, L. Huang, and K. Church, «Speech emotion recognition with multi-task learning», vol. 1, 2021. DOI: 10.21437/Interspeech.2021-1852.

[3] F. Z. Canal, T. R. Müller, J. C. Matias, *et al.*, «A survey on facial emotion recognition techniques: A state-of-the-art literature review», *Information Sciences*, vol. 582, 2022, ISSN: 00200255. DOI: 10.1016/j.ins.2021.10.005.

[4] S. Chan, *Zoom remained among q2 2021's most downloaded apps with more than 900 million installs to date*, Jul. 2021. [Online]. Available: https://sensortower.com/blog/zoom-top-apps-q2-2021 (visited on 01/10/2023).

[5] B. Sredojev, D. Samardzija, and D. Posarac, «Webrtc technology overview and signaling solution design and implementation», Institute of Electrical and Electronics Engineers Inc., Jul. 2015, pp. 1006–1009, ISBN: 9789532330854. DOI: 10.1109/MIPRO.2015.7160422.

[6] C. Jian and Z. Lin, «Research and implementation of webrtc signaling via websocket-based for real-time multimedia communications», 2016.

[7] G. Suciu, S. Stefanescu, C. Beceanu, and M. Ceaparu, «Webrtc role in real-time communication and video conferencing», 2020. DOI: 10.1109/GIOTS49054.2020.9119656.

[8] R. Sharma, *Global webrtc market 2022 top industry trend and segments analysis up to 2028*, Jul. 2022. [Online]. Available: https://www.linkedin.com/pulse/global-webrtc-market-2022-top-industry-trend-segments-rivana-sharma (visited on 01/12/2023).

[9] *What is webrtc?* [Online]. Available: https://trueconf.com/webrtc.html (visited on 01/19/2023).

[10] *What is webrtc and what is it good for?*, Oct. 2022. [Online]. Available: https://bloggeek.me/what-is-webrtc/ (visited on 01/11/2023).

[11] A. B., *Environment: Signaling, stun and turn servers*. [Online]. Available: https://support.medialooks.com/hc/en-us/articles/360000213312-Environment-signaling-STUN-and-TURN-servers (visited on 01/19/2023).

[12] S. Lööf and S. Holm, «The design and architecture of a webrtc application», *IEEE Cloud Computing*, vol. 3, 5 2016, ISSN: 23256095.

[13] K. Honney, *Webrtc servers and multi-party communication in webrtc*, Jan. 2019. [Online]. Available: https://medium.com/@khan_honney/webrtc-servers-and-multi-party-communication-in-webrtc-6bf3870b15eb.

[14] S. Petrangeli, D. Pauwels, J. van der Hooft, *et al.*, «A scalable webrtc-based framework for remote video collaboration applications», *Multimedia Tools and Applications*, vol. 78, pp. 7419–7452, 6 Mar. 2019, ISSN: 15737721. DOI: 10.1007/s11042-018-6460-0.

[15] B. Grozev, *Improving scale and media quality with cascading sfus (boris grozev)*, Nov. 2018.

[16] R. Sasson, *How vidyo.io delivers massive scalability while maintaining reliability and quality through cascading sfus*, Dec. 2017. [Online]. Available: https://vidyo.io/blog/features/vidyo-io-delivers-massive-scalability-maintaining-reliability-quality-cascading-sfus/.

[17] Zoom, *Overview core concepts and components - zoom*. [Online]. Available: https://explore.zoom.us/docs/doc/Zoom%20Connection%20Process%20Whitepaper.pdf.

[18] C. Cressler, *A study of zoom's video conferencing architecture amp; system design*, Aug. 2021. [Online]. Available: https://www.cometchat.com/blog/zoom-video-technology-architecture.

[19] L. Vivien, *How zoom works – architecture illustrated*, May 2022. [Online]. Available: https://www.lavivienpost.com/how-zoom-works.

[20] Z. V. C. Inc., *Zoom architected for reliability*, Sep. 2019. [Online]. Available: https://zoomgov.com/docs/doc/Zoom_Global_Infrastructure.pdf (visited on 01/19/2023).

[21] Y. Chen, *Why zoom's success is not a coincidence — on distributed video conferencing architecture*. [Online]. Available: https://www.linkedin.com/pulse/why-zooms-success-coincidence-distributed-video-yiheng-intel-chen?trk=public_profile_article_view.

[22] N. Walia, *Here's how zoom provides industry-leading video capacity*, Jun. 2019. [Online]. Available: https://blog.zoom.us/zoom-can-provide-increase-industry-leading-video-capacity/.

[23] V. Sachdeva, *Zoom — video conf app at scale*, May 2020. [Online]. Available: https://medium.com/@vsachdeva/zoom-video-conf-tool-at-scale-e86289c290b8.

[24] J. Vass, *How discord handles two and half million concurrent voice users using webrtc*, Sep. 2018. [Online]. Available: https://discord.com/blog/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc.

[25] *Welcome to kurento*. [Online]. Available: https://doc-kurento.readthedocs.io/.

[26] L. López, B. García, R. Benítez, *et al.*, «Kurento: The webrtc modular media server», Association for Computing Machinery, Inc, Oct. 2016, pp. 1187–1191, ISBN: 9781450336031. DOI: 10.1145/2964284.2973798.

[27] B. Garcia, L. Lopez-Fernandez, M. Gallego, and F. Gortazar, «Kurento: The swiss army knife of webrtc media servers», *IEEE Communications Standards Magazine*, vol. 1, pp. 44–51, 2 2017, ISSN: 24712825. DOI: 10.1109/MCOMSTD.2017.1700006.

[28] B. García, L. López, F. Gortázar, M. Gallego, and G. A. Carella, «Nubomedia: The first open source webrtc paas», Association for Computing Machinery, Inc, Oct. 2017, pp. 1205–1208, ISBN: 9781450349062. DOI: 10.1145/3123266.3129392.

[29] M. Maithri, U. Raghavendra, A. Gudigar, *et al.*, «Automated emotion recognition: Current trends and future perspectives», *Computer Methods and Programs in Biomedicine*, vol. 215, p. 106 646, Mar. 2022, ISSN: 0169-2607. DOI: 10.1016/J.CMPB.2022.106646.

[30] Mustaqeem and S. Kwon, «Mlt-dnet: Speech emotion recognition using 1d dilated cnn based on multi-learning trick approach», *Expert Systems with Applications*, vol. 167, 2021, ISSN: 09574174. DOI: 10.1016/j.eswa.2020.114177.

[31] S. Parthasarathy and C. Busso, «Semi-supervised speech emotion recognition with ladder networks», *IEEE/ACM Transactions on Audio Speech and Language Processing*, vol. 28, 2020, ISSN: 23299304. DOI: 10.1109/TASLP.2020.3023632.

[32] Z. Zhao, Q. Li, Z. Zhang, *et al.*, «Combining a parallel 2d cnn with a self-attention dilated residual network for ctc-based discrete speech emotion recognition», *Neural Networks*, vol. 141, 2021, ISSN: 18792782. DOI: 10.1016/j.neunet.2021.03.013.

[33] M. Farooq, F. Hussain, N. K. Baloch, F. R. Raja, H. Yu, and Y. B. Zikria, «Impact of feature selection algorithm on speech emotion recognition using deep convolutional neural network», *Sensors (Switzerland)*, vol. 20, 21 2020, ISSN: 14248220. DOI: 10.3390/s20216008.

[34] Z. Yang and Y. Huang, «Algorithm for speech emotion recognition classification based on mel-frequency cepstral coefficients and broad learning system», *Evolutionary Intelligence*, vol. 15, 4 2022, ISSN: 18645917. DOI: 10.1007/s12065-020-00532-3.

[35] P. Naga, S. D. Marri, and R. Borreo, «Facial emotion recognition methods, datasets and technologies: A literature survey», *Materials Today: Proceedings*, vol. 80, 2023, ISSN: 22147853. DOI: 10.1016/j.matpr.2021.07.046.

[36] D. Y. Choi and B. C. Song, «Semi-supervised learning for facial expression-based emotion recognition in the continuous domain», *Multimedia Tools and Applications*, vol. 79, 37-38 2020, ISSN: 15737721. DOI: 10.1007/s11042-020-09412-5.

[37] M. K. Chowdary, T. N. Nguyen, and D. J. Hemanth, «Deep learning-based facial emotion recognition for human–computer interaction applications», *Neural Computing and Applications*, 2021, ISSN: 14333058. DOI: 10.1007/s00521-021-06012-8.

[38] A. I. Argesanu and G. D. Andreescu, «A platform to manage the end-to-end lifecycle of batch-prediction machine learning models», Institute of Electrical and Electronics Engineers Inc., May 2021, pp. 329–334, ISBN: 9781728195445. DOI: 10.1109/SACI51354.2021.9465588.

[39] S. Tang, B. He, C. Yu, Y. Li, and K. Li, «A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications», *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, pp. 71–91, 1 Jan. 2022, ISSN: 15582191. DOI: 10.1109/TKDE.2020.2975652.

[40] «A scalable machine learning online service for big data real-time analysis», Uses spark for real-time machine learning serving., Institute of Electrical and Electronics Engineers Inc., Jan. 2015, ISBN: 9781479945412. DOI: 10.1109/CIBD.2014.7011537.

[41] D. Kılınç, «A spark-based big data analysis framework for real-time sentiment prediction on streaming data», *Software: Practice and Experience*, vol. 49, pp. 1352–1364, 9 Sep. 2019, ISSN: 0038-0644. DOI: 10.1002/spe.2724.

[42] S. Horchidan, E. Kritharakis, V. Kalavri, and P. Carbone, «Evaluating model serving strategies over streaming data», Association for Computing Machinery, Inc, Jun. 2022, ISBN: 9781450393751. DOI: 10.1145/3533028.3533308.

[43] *Minimizing real-time prediction serving latency in machine learning*. [Online]. Available: https://cloud.google.com/architecture/minimizing-predictive-serving-latency-in-machine-learning.

[44] D. Kreuzberger, N. Kühl, and S. Hirschl, «Machine learning operations (mlops): Overview, definition, and architecture».

[45] A. Banerjee, C.-C. Chen, C.-C. Hung, X. Huang, Y. Wang, and R. Chevesaran, *Challenges and Experiences with MLOps for Performance Diagnostics in Hybrid-Cloud Enterprise Software Deployments*, ISBN: 9781939133151. [Online]. Available: https://www.usenix.org/conference/opml20/presentation/banerjee.

[46] I. Karamitsos, S. Albarhami, and C. Apostolopoulos, «Applying devops practices of continuous automation for machine learning», *Information (Switzerland)*, vol. 11, pp. 1–15, 7 Jul. 2020, ISSN: 20782489. DOI: 10.3390/info11070363.

[47] A. Posoldova, «Machine learning pipelines: From research to production», *IEEE Potentials*, vol. 39, pp. 38–42, 6 Nov. 2020, ISSN: 15581772. DOI: 10.1109/MPOT.2020.3016280.

[48] S. Mei, C. Liu, Q. Wang, and H. Su, «Model provenance management in mlops pipeline», Association for Computing Machinery, Jan. 2022, pp. 45–50, ISBN: 9781450395717. DOI: 10.1145/3512850.3512861.

[49] I. Kumara, D. D. Nucci, W. J. V. Den, and D. Andrew, «Requirements and reference architecture for mlops:insights requirements and reference architecture for mlops:insights from industry from industry», 2022. DOI: 10.36227/techrxiv.21397413.v1. [Online]. Available: https://doi.org/10.36227/techrxiv.21397413.v1.

[50] L. Baier and S. Seebacher, «Challenges in the deployment and operation of machine learning in practice».

[51] P. Ruf, M. Madan, C. Reich, and D. Ould-Abdeslam, «Demystifying mlops and presenting a recipe for the selection of open-source tools», *Applied Sciences (Switzerland)*, vol. 11, 19 Oct. 2021, ISSN: 20763417. DOI: 10.3390/app11198861.

[52] *Mlops principles*. [Online]. Available: https://ml-ops.org/content/mlops-principles (visited on 01/17/2023).

[53] P. Klushin, *What does it take to deploy ml models in production?*, Apr. 2022. [Online]. Available: https://www.qwak.com/post/what-does-it-take-to-deploy-ml-models-in-production (visited on 01/12/2023).

[54] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, *et al.*, «Serverless computing: One step forward, two steps back», Dec. 2018. [Online]. Available: http://arxiv.org/abs/1812.03651.

[55] V. Ishakian, V. Muthusamy, and A. Slominski, «Serving deep learning models in a serverless platform», Institute of Electrical and Electronics Engineers Inc., May 2018, pp. 257–262, ISBN: 9781538650080. DOI: 10.1109/IC2E.2018.00052.

[56] S. I. Serengil and A. Ozpinar, «Lightface: A hybrid deep face recognition framework», in *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)*, IEEE, 2020, pp. 23–27. DOI: 10.1109/ASYU50717.2020.9259802. [Online]. Available: https://doi.org/10.1109/ASYU50717.2020.9259802.

[57] S. I. Serengil and A. Ozpinar, «Hyperextended lightface: A facial attribute analysis framework», in *2021 International Conference on Engineering and Emerging Technologies (ICEET)*, IEEE, 2021, pp. 1–4. DOI: 10.1109/ICEET53442.2021.9659697. [Online]. Available: https://doi.org/10.1109/ICEET53442.2021.9659697.

[58] K. Wang, Q. Wu, L. Song, *et al.*, «Mead: A large-scale audio-visual dataset for emotional talking-face generation», in *ECCV*, 2020.

[59] M. G. de Pinto, M. Polignano, P. Lops, and G. Semeraro, «Emotions understanding model from spoken language using deep neural networks and mel-frequency cepstral coefficients», in *2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS)*, 2020, pp. 1–5.