UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO DE FIN DE MÁSTER

DESARROLLO DE UNA HERRAMIENTA DE BÚSQUEDA MASIVA DE ACTIVOS DIGITALES BASADA EN REPRESENTACIONES DE MODALIDAD CRUZADA

PABLO REGODÓN CEREZO

FEBRERO 2025

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Trabajo de Fin de Máster

Título: Desarrollo de una herramienta de búsqueda masiva de activos digitales basada en representaciones de modalidad cruzada.

Autor: D. Pablo Regodón Cerezo.

Tutor: D. José Luis Blanco Murillo.

Cotutor: D. Juan Gutiérrez Navarro.

Departamento: Señales, Sistemas y Radiocomunicaciones.

Miembros del tribunal

Presidente/a: D./D.ª

Vocal: D./D.ª

Secretario/a: D./D.ª

Suplente: D./D.ª

Los miembros del tribunal arriba nombrados acuerdan otorgar la calificación de: .

Madrid, a de de 20 .

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO DE FIN DE MÁSTER

DESARROLLO DE UNA HERRAMIENTA DE BÚSQUEDA MASIVA DE ACTIVOS DIGITALES BASADA EN REPRESENTACIONES DE MODALIDAD CRUZADA

PABLO REGODÓN CEREZO

FEBRERO 2025

" Cuando yo salí del pueblo, hace la friolera de cuarenta y ocho años, me topé con el Aniano, «el Cosario», bajo el chopo del Elicio, frente al palomar de la tía Zenona, ya en el camino de Pozal de la Culebra. Y el Aniano se vino a mí y me dijo: «¿Dónde va el Estudiante?». Y yo le dije: «¡Qué sé yo! Lejos». «¿Por tiempo?», dijo él. Y yo le dije: «Ni lo sé». Y él me dijo con su servicial docilidad: «Voy a la capital. ¿Te se ofrece algo?». Y yo le dije: «Nada, gracias, Aniano». "

— *Viejas historias de Castilla la Vieja*, **Miguel Delibes**.

Abstract

Efficient management and storage of digital assets are key challenges in Artificial Intelligence. Despite advancements in handling and processing tasks, few contributions have addressed AI applications for massive management of those records and information flows. However, there is a more prominent need to improve data management and develop and train new models every day. This task requires reducing the processing and storage effort while accelerating the internal processes and maximizing resource performance on search procedures.

Database systems have been experimenting with continuous advancements in recent years, including new ingestion, digestion, writing, reading, and retrieving capabilities. The last revolution brought noSQL schemas, which eased data management and handling, avoiding the explicit definition of relations and promoting the use of minimally or fully unstructured data. noSQL schemas demand different queries, employing new comparison and result-picking criteria. The challenge of defining and using these languages has led to ongoing efforts to look for SQL-compatible or SQL-based languages to expedite development.

Deep representations guided by Artificial Intelligence models are changing how we interact with data. Predefined size-vector representations (embeddings) linked to cross-modal models (e.g., text and image) and handling multiple indexes allow new semantic search schemas. It is possible to perform massive searches by exploiting the topology and geometry that underlay these representations and the generative capabilities of the underlying models. Moreover, for these to be performed efficiently, optimizing the searching approach by adjusting the structure and characteristics of the algebraic or topological space as conveniently as possible is necessary.

Video surveillance systems (CCTV, Closed-Circuit Television) must cope with large amounts of unstructured multimedia data. An efficient way of handling this management is to minimize the raw data operations whenever possible, using their corresponding vector representations.

This work describes the design, development, and assessment of an advanced massive search system using cross-modal representations. It focused on optimizing the processes of ingestion, digestion, content retrieval, and summary generation in the context of a video surveillance system employing cross-modal deep representations of multimedia data.

We focused on the datasets of UCF-Crime (short videos) and UCF Annotation (UCA, timestamps and event description). We ingested these into the databases using multimodal representations and implemented an efficient pipeline for retrieval. By comparing the two technologies, we conclude that database implementations are complex but can be used to ease AI training and assessment. Our results evidence the solution's good performance in terms of accuracy and speed in the retrieval task.

Resumen

La gestión y almacenamiento eficientes de activos digitales son desafíos clave en la Inteligencia Artificial. A pesar de los avances en el manejo y procesamiento de tareas, pocas contribuciones han abordado aplicaciones de IA para la gestión masiva de esos registros y flujos de información. Sin embargo, cada día hay una necesidad más prominente de mejorar la gestión de datos y desarrollar y entrenar nuevos modelos. Esta tarea requiere reducir el esfuerzo de procesamiento y almacenamiento mientras se aceleran los procesos

internos y se maximiza el rendimiento de los recursos en los procedimientos de búsqueda.

Los sistemas de bases de datos han experimentado avances continuos en los últimos años, incluyendo nuevas capacidades de ingesta, digestión, escritura, lectura y recuperación. La última revolución trajo los esquemas noSQL, que facilitaron la gestión y manejo de datos, evitando la definición explícita de relaciones y promoviendo el uso de datos mínimamente o totalmente desestructurados. Los esquemas noSQL demandan consultas diferentes, empleando nuevos criterios de comparación y selección de resultados. El desafío de definir y usar estos lenguajes ha llevado a esfuerzos continuos para buscar lenguajes compatibles con SQL o basados en SQL para acelerar el desarrollo.

Las representaciones profundas guiadas por modelos de Inteligencia Artificial están cambiando la forma en que interactuamos con los datos. Las representaciones vectoriales de tamaño predefinido (embeddings) vinculadas a modelos multimodales (por ejemplo, texto e imagen) y el manejo de múltiples índices permiten nuevos esquemas de búsqueda semántica. Es posible realizar búsquedas masivas explotando la topología y geometría que subyacen a estas representaciones y las capacidades generativas de los modelos subyacentes. Además, para que estas se realicen de manera eficiente, es necesario optimizar el enfoque de búsqueda ajustando la estructura y características del espacio algebraico o topológico de la manera más conveniente posible.

Los sistemas de videovigilancia (CCTV, Circuito Cerrado de Televisión) deben lidiar con grandes cantidades de datos multimedia no estructurados. Una forma eficiente de manejar esta gestión es minimizar las operaciones de datos en bruto siempre que sea posible, utilizando sus correspondientes representaciones vectoriales.

Este trabajo describe el diseño, desarrollo y evaluación de un sistema avanzado de búsqueda masiva utilizando representaciones multimodales. Se centró en optimizar los procesos de ingesta, digestión, recuperación de contenido y generación de resúmenes en el contexto de un sistema de videovigilancia empleando representaciones profundas multimodales de datos multimedia.

Nos enfocamos en los conjuntos de datos de UCF-Crime (videos cortos) y UCF Annotation (UCA, marcas de tiempo y descripción de eventos). Los ingerimos en las bases de datos utilizando representaciones multimodales e implementamos un pipeline eficiente para la recuperación. Al comparar las dos tecnologías, concluimos que las implementaciones de bases de datos son complejas pero pueden usarse para facilitar el entrenamiento y evaluación de IA. Nuestros resultados evidencian el buen rendimiento de la solución en términos de precisión y velocidad en la tarea de recuperación.

Keywords

Artificial Intelligence, Cross-modal representations, Unstructured data, Semantic search, Video surveillance, Content retrieval, Content Summarization, Search optimization.

Palabras clave

Inteligencia Artificial, Representaciones de modalidad cruzada, Datos no estructurados, Búsqueda semántica, Videovigilancia, Recuperación de contenido, Resumen de contenido, Optimización de búsqueda.

Contents

List of Figures

List of Tables

Listings

Acronyms

ACID Atomicity, Consistency, Isolation and Durability. 45

AI Artificial Intelligence. 3, 73

ANN Approximate Nearest Neighbors. 41

ANNOY Approximate Nearest Neighbors Oh Yeah. 19, 41

API Application Programming Interface. 36, 44, 50–52, 54, 55, 57, 74

CAGR Compound Annual Growth Rate. 20

CCTV Closed-Circuit Television. v–vii, 2, 5, 6, 8, 23, 24, 26, 31, 33, 49, 64, 68, 73–75

CLAP Contrastive Language-Audio Pretraining. 75

CLIP Contrastive Language-Image Pre-training. 2, 13–16, 34, 38–41, 46, 54, 56, 61, 68

CNN Convolutional Neural Network. 11, 13, 16

CPU Central Processing Unit. vi, 41

CSS Cascading Style Sheets. 51, 56

DL Deep Learning. 10, 11, 23, 74

FFMPEG Fast Forward Moving Picture Experts Group. 53

FTP File Transfer Protocol. 32, 33, 38, 53

GAPS *Grupo de Aplicaciones del Procesado de Señal*, Signal Processing Applications Group. 68

GPT Generative Pre-training Transformer. 16

GPU Graphics Processing Unit. 41

HNSW Hierarchical Navigable Small Worlds. 18, 20, 22, 41, 43–45, 67

HTML Hypertext Mark-up Language. 51, 56

HTTP Hypertext Transfer Protocol. 35, 49, 51–54

IVF Inverted File. 18, 20, 22, 41–43, 45

JSON JavaScript Object Notation. 49, 53, 54, 58

NLP Natural Language Processing. 13, 14

PASE PostgreSQL Approximate nearest neighbor Search Extension. 22, 45, 46

PTZ Pan, Tilt, Zoom. 6

REST Representational State Transfer. 49–52, 54, 57

RTP Real-time Transport Protocol. 7

SIMD Single Instruction, Multiple Data. 43

SQL Structured Query Language. 22, 45, 47

SVM Support Vector Machine. 9

UCA UCF-Crime Annotation. 26, 27, 32, 38–44, 49, 52, 53, 56, 58, 63, 64, 66, 67

UCF University of Central Florida. 23, 25–27, 29, 35, 38–41, 43, 44, 49, 50, 52, 53, 56, 58, 62–64, 66–68, 75

URI Unified Resource Identifier. 49, 51, 57, 58, 68

USD United States Dollar. 20

VDBMS Vector Database Management System. 16, 17, 20, 22, 44, 74, 75

ViT Vision Transformer. 14, 16

Chapter 1

Introduction

1.1 Overview

In digital surveillance and security, the ability to efficiently search and analyze vast amounts of visual data is crucial. This Master's Thesis addresses the pressing need for advanced technologies in processing and retrieving information from video surveillance footage. The exponential growth of surveillance cameras in public and private spaces has resulted in an overwhelming volume of video data. Traditional methods of searching through this data are often time-consuming and inefficient, particularly when locating specific incidents or individuals across multiple video sources.

This research aims to develop a tool that leverages cross-modal representations to enhance search capabilities for digital assets, specifically images extracted from video surveillance footage. We seek to create a robust system capable of understanding and correlating visual and textual information using advanced computer vision techniques, natural language processing, and machine learning algorithms. The proposed tool will enable users to perform complex queries across large datasets of surveillance imagery, significantly reducing the time and resources required for video analysis in security and law enforcement applications.

This thesis explores the theoretical foundations, practical implementation, and ethical considerations of such a system to contribute to intelligent video surveillance and set a new standard for efficient digital asset search in the context of public safety and security.

1.2 Motivation

According to IHS Markit's 2019 prediction, by the end of 2021 over one billion surveillance cameras should have been installed [1]. This projection has been widely cited and used as a benchmark in subsequent studies and reports. China leads in surveillance camera deployment, with estimates ranging from 54% to 60% of the global total. Based on these figures, China is estimated to have between 540 million and 626 million cameras installed. These numbers translate to approximately 372.8 to 439.07 cameras per 1,000 people, significantly higher than other countries. In contrast, other major cities have much lower camera densities: Tokyo has 1.06 cameras per 1,000 people, Mexico City 3.62, New York

6.87, and Los Angeles 8.77 [2].

The global video surveillance market has seen substantial growth, reaching $9.72 billion for home video surveillance cameras in 2023, showing a 15% increase from $8.43 billion in 2022. The market is expected to grow at a compound annual growth rate of 8.04%, potentially reaching $12.3 billion by 2032 [1]. It is worth noting that despite the increasing prevalence of surveillance cameras, to this day, studies have found little correlation between the number of public CCTV cameras and crime or safety rates [3].

The integration of cross-modal representations offers a promising avenue for enhancing the capabilities of digital asset search tools. Cross-modality refers to the ability to process and analyze information across different sensory modalities or data types. In the context of surveillance and security, this approach can combine visual data from cameras with other forms of information, such as audio, text, or sensor data. By leveraging cross-modal representations, the proposed search tool can provide a more comprehensive and nuanced understanding of surveillance footage. For example, it could correlate visual cues with audio events or textual descriptions, enabling more sophisticated queries and improving the accuracy of search results. Integrating multiple modalities can significantly enhance the tool's ability to detect and analyze complex scenarios, ultimately leading to more effective security measures and faster response times in critical situations.

A way to expedite effective footage review by human agents is to use cross-modal capabilities to summarize surveillance video footage. The tool may provide a selection of video clips that summarize the footage attending to an input text query.

1.3 Objectives

The main objective addressed in this Master Thesis is to develop a retrieval tool for video footage driven by the use of cross-modal representations. To achieve this, the capabilities of a widely spread model for visual and textual encoding known as CLIP will be studied, as well as the use of database management systems for the vector data this model outputs. Then, a full application will be implemented, deployed and tested.

To cover this main goal, we identified the following secondary objectives and integrated them into our methodology. By covering these secondary goals, we aim to satisfy the main one.

1. Familiarization with cross-modal representations or embeddings.

2. Familiarization with vector database systems and comparative study of different implementations.

3. Deployment of index structures to search vector databases.

4. Formulation and completion of data ingestion benchmarks.

5. Formulation and completion of data retrieval benchmarks.

1.4 Methodology

This thesis required previous research on cross-modality technologies, search engines, and tools; as well as massive data asset management. The described work adopted an applied research approach that combined quantitative and qualitative methods to develop and evaluate the search tool.

The development process was organized into one-week sprints starting on Tuesdays, each with the following elements. The sprint began with the Sprint Planning with the supervisor, where goals and tasks for the week ahead were defined. Midway through the week, a brief Mid-Week Check-in was held to discuss progress and address any blockers. At the end of the sprint, a Sprint Review takes place, involving a demonstration of completed. Each sprint concluded with a Sprint Retrospective and Planning for the next sprint.

Hereafter, we summarize the stages covered in the same order they were conducted:

1. **Research stage**: Focusing on cross-modality AI and retrieval:
2. **Implementation stage**: Addressing the implementation of cross-modal capabilities for retrieval. On this occasion, we shall not focus on the data ingestion process but on the retrieval pipeline, including the elements that ease it.
3. **Evaluation stage**: Based on the results obtained from the completed implementation and tests. We consider both the retrieval results and the system's experience based on the time-to-response of the retrieval.
4. **Ethical considerations and Impacts**. Finally, the ethical aspects of the address activities were analyzed in depth. This process included both the technologies involved and the solution proposed. The impact of the work was also assessed, including socio-economic and environmental aspects, along with the ethics already mentioned.

1.5 Structure of this document

This document is divided in the following chapters:

- **Chapter 1, Introduction**: in this chapter, a brief introduction to the work performed through this Master Thesis is given, listing the main objectives and describing the structure of the present document.
- **Chapter 2, Background**: in this chapter, a top-down approach to the subjects studied in the work is presented.
- **Chapter 3, Implementation**: in this chapter, a bottom-up approach to the developments of the work is presented, from the low-level data management to the final application.
- **Chapter 4, Results**: in this chapter, a study of the main outcomes of this work is presented.

- **Chapter 5, Conclusion**: in this chapter, a review of the outcomes of this Master Thesis is presented, and some future lines of work are proposed.

Finally, Appendices provide additional information on the work conducted, including the analysis of the impact of the work conducted, Annex A, and a summary of the project budget, Annex B.

Chapter 2

Background

This chapter summarizes the technical background and the main topics covered in this Master Thesis, from the use case to be addressed to the data representations and the tools to cover the retrieval service.

Section 2.1 analyzes video surveillance solutions, the primary use case in this work, as it is a relevant example of a system addressing digital asset creation and management. Section 2.2 exposes the different techniques and approaches of video summarization and key events retrieval. Section 2.3 discusses techniques to transform multimedia data into vector representations, paying special attention to transformations that can be applied to different types of data (text, image, video, etc.) and maintain semantic coherence. Finally, Section 2.4 covers some of the characteristics of the databases and data management tools that can be used to store these vector representations of multimedia data, along with some actual implementations available on the market.

2.1 Video surveillance

According to [4], video surveillance is the «action of monitoring the activity in public or private scenes using cameras». Also known as CCTV, video surveillance needs to manage considerable amounts of multimedia content, especially visual data[1], demanding solutions with fast performance and reliability. These CCTV systems have experienced massive growth in the last decades thanks to technological advances that helped introduce better and cheaper equipment. These advancements coexist with an increasing need for safer environments in public and private spaces, mainly spread after the 9/11 attacks in the World Trade Center.

Regarding the historical context of video surveillance, early CCTV solutions were deployed in different cities during the 1960s to provide police departments with a monitoring tool for public spaces. During the '60s and '70s, these primitive systems spread and gained functionalities and experienced a massive boom in the '80s thanks to solid-state electronics, which made components more reliable, smaller, and cheaper. The '90s and '2000s witnessed the transition from analog to digital technology, exploiting elements such

[1]Audio data is also an information source in video surveillance; however, in this work, only visual content is considered since it is the only modality most CCTV systems use.

as Internet communications, extensive storage infrastructure, and advanced video coding schemes [5].

Throughout the following years, the more advanced CCTV systems began to care not only about video acquisition but introduced video analysis to automatically extract metadata or valuable information. However, this task remains unsolved [6].

Figure 2.1 presents the main functionalities of a video surveillance system, as stated by Elharrouss *et al.* in [4]. *Acquisition* involves retrieving inputs such as images from the cameras or other input devices (such as presence sensors, etc.). *Recording* encompasses storing the video content for later use. *PTZ control* works for the position of the cameras. *Visualization* determines how users can inspect the acquired content, including (1) how the video is presented to the user as well as (2) the process of retrieving the visual content from the video storage. *Alarm* involves sending out a notification to the user whenever an important event occurs in the acquisition environment. Finally, all these functionalities are put to work through a switching (networking) and processing infrastructure.

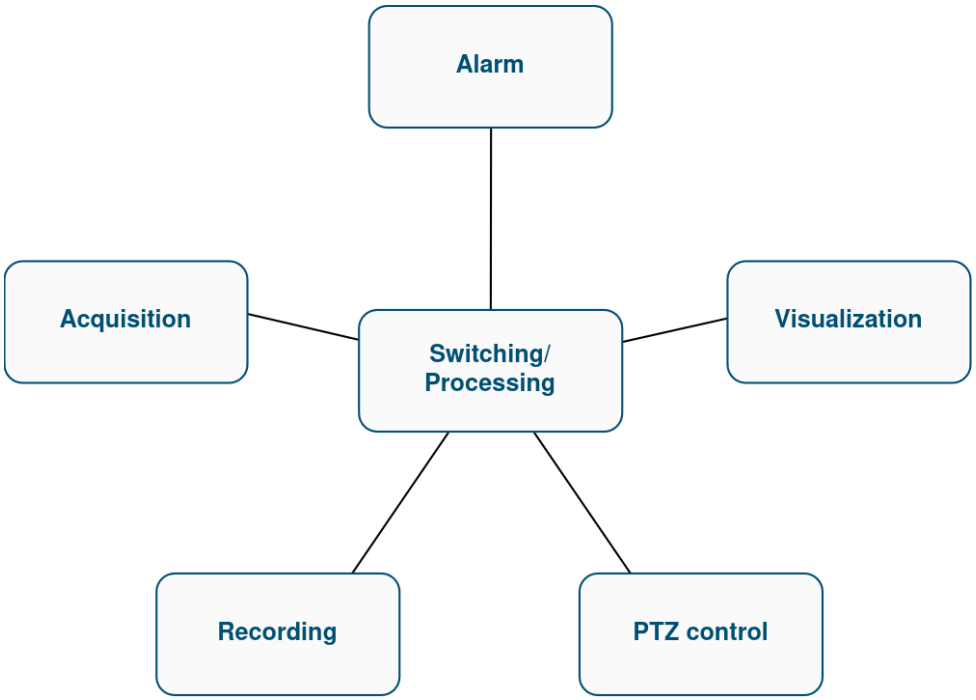

Figure 2.1: Main functionalities of a video surveillance system. Extracted from [4].

The article by Elharrouss *et al.* also introduces the division of video surveillance system design into architectural design (involving the types of cameras, the transmission means, the servers used to process the data, etc.) and analysis design. The latter is of special importance to this Thesis since it is through analysis that the final functionalities are obtained.

Figure 2.2 illustrates some analysis tasks that can be performed over a video surveillance architecture. The video summarization is studied in this project, and a deeper insight is given in Section 2.2.

As for the architectural part, we used the Cisco Video Surveillance Solution [7] as the primary reference. As can be seen in Figure 2.3, the main components involved in the architecture are video endpoints (cameras), client endpoints, which include the

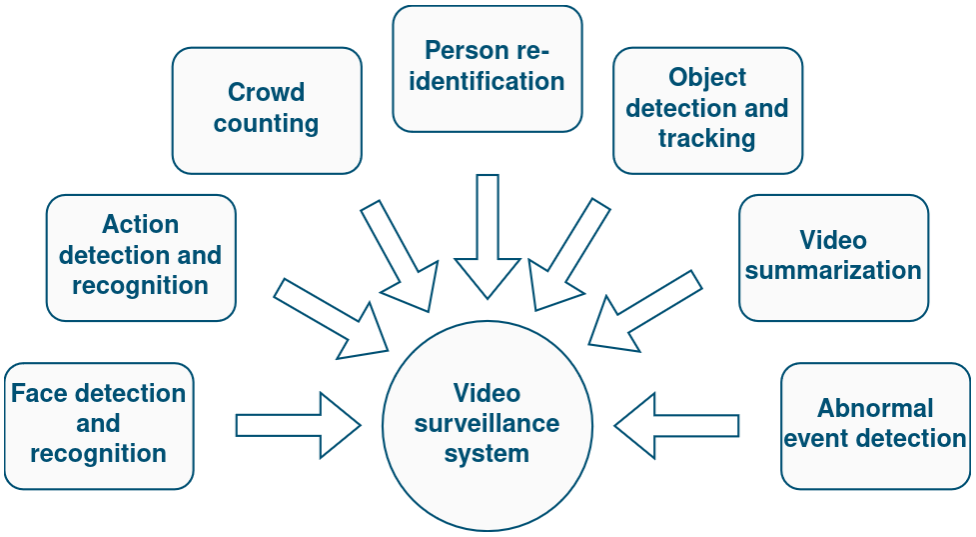

Figure 2.2: Main analysis topics related to video surveillance systems. Extracted from [4].

application through which clients interact with the video surveillance system, and video surveillance manager, which includes the whole computing infrastructure needed to provide the functionalities. These three components are connected by a shared IP network, whose scope is flexible and depends on the client's requirements.

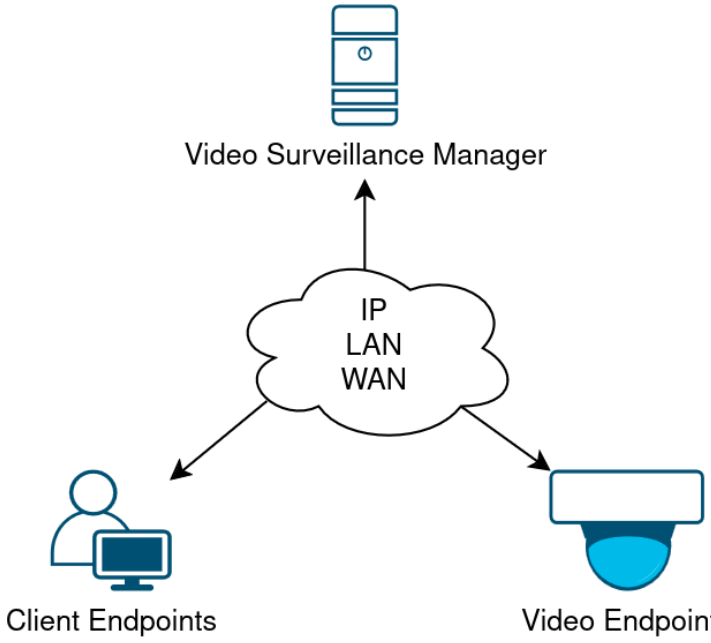

Figure 2.3: Components of Cisco CCTV solution. Extracted from [7].

Figure 2.4 illustrates the usual information exchange between the solution's components. Video endpoints continuously stream content to a media server through a network infrastructure and a video transmission protocol, such as RTP. This media server acts as a proxy, establishing sessions with client endpoints for video delivery. Clients can also interact with the operations manager. The latter is a control component that can change the media server's settings, among other things. Note that not all the possible functionalities of the media server are presented in the figure. The solution can be (and is) in charge of video storage, processing, and delivering video content to analysis servers,

among other functionalities not illustrated.

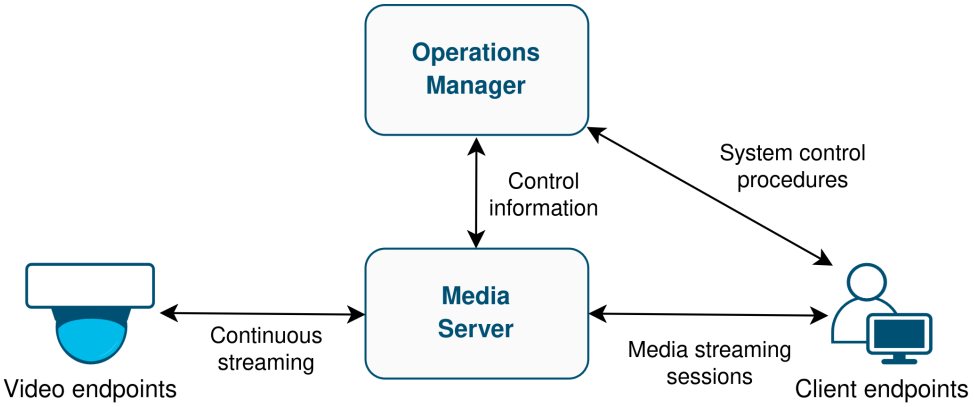

Figure 2.4: Data flow of Cisco CCTV solution. Extracted from [7].

Cisco CCTV solution does not cover in detail the physical implementation of the components shown in Figure 2.3, just mentions some requirements the hardware must met: Video Endpoints must have IP communications capabilities, and Client Endpoints must be able to cope with video decoding. Video Surveillance Manager server applications can be deployed in Linux-based Multi-Service Platforms (based on Red-Hat or SUSE, for example) or in Virtual Machines running on top of generic servers.

In this Thesis, the physical side of a CCTV system will not be covered, since these deployments are usually client-specific and present a number of parameters which can broadly affect the design (number of cameras, privacy concerns, long-term storage capabilities, etc.).

This section has described the general functionalities and technologies of a CCTV system. From this point forward, the Thesis will focus on the functionalities that have been studied.

2.2 Video summarization

One of the most critical bottlenecks in video surveillance is manually reviewing video recordings to look for relevant events [4]. Video summarization is the approach to automating this work we aim to discuss.

Video summarization comprises a set of automatic techniques that can identify meaningful or prominent information from videos. The goal is to ease retrieval, as we have just commented, as well as storage, indexing, quick browsing, and sharing [8]. Two basic types of video summarization exist: retrieving the key frames from a video (static summary) and retrieving the key shots from a video (dynamic summary).

Regarding terminology, [8] defines the following terms that will be recurrently used from here on throughout this Thesis:

- **Scene** (S): a physical environment of interest to be captured optically.
- **View** (V): a perspective of the scene S from which the underlying optical information is captured.

- **Shot** (s): an uninterrupted sequence of frames captured from a given view, V. The transition from two different shots is called a cut. The frames in a shot tend to be homogeneous; that is, they share visual and semantic information. The information required for this homogeneity depends on a user-specified threshold, T.

- **Frame** (f): an image of a scene, S, captured at a particular instance, t.

- **Event** (e): an event e at a given time t denotes a significant change in the underlying information between two consecutive frames, (f_t, f_{t+1}), to break homogeneity (i.e., end of a shot s). Events in a given video can be represented in a collection of frames with significant change concerning selected metric function D and threshold T: $e := \{f_t \mid D(f_t - f_{t-1}) > T\}$.

- **Key-frame** (k): a frame f that best represents the content of a shot s.

Machine learning techniques are widely used to generate the desired summaries in video summarization. These techniques involve an initial feature extraction step on the video frames to perform the summary generation subsequentially through some mathematical algorithm (e.g., K-means, random forest, logistic regression, etc.) performed over those features. The feature extraction step aims to transform the video content into a structured and numerical representation, known as a feature vector, simplifying and compressing the video. The goal is to create a representation that captures the semantic meaning of the content through its features, thus easing the task of finding meaningful or relevant elements for the summarization algorithms.

Classical feature extraction is known to suffer from severe limitations. It worked with "handcrafted" features like the color histogram or optical flow [9], which do not assure that the features reflect the semantic meaning of the content. For instance, the extraction of a frame's color histogram does not accurately represent what is in the image: the histogram of a milk cow and a Dalmatian dog are prone to be similar but represent different things. Moreover, anticipating which features are helpful for the task is challenging. Intensive experimentation is usually performed until acceptable outcomes are obtained, making this process tedious and time consuming.

For the summarization step, classical machine learning techniques are applied to the feature vectors to obtain the desired summaries. K-means clustering and SVMs are examples of techniques for this task [8]. The usual approach is to test several of these algorithms, combining different sets of hyperparameters, and selecting the best based on the performance on the training and evaluation sets.

Overall, the main flaw of the classical approach for video summarization is the dependence on the feature extraction step, which is quite restrictive. Nevertheless, acceptable results can be obtained with these algorithms for more straightforward content, and they have the advantage of being faster than their deep learning counterparts.

The following section presents a more detailed exploration of the deep learning approaches for feature extraction and video summarization.

2.3 Cross-modal representations

Multimedia data is a form of unstructured data, that is, information with redundancy and no internal structure that is known *a priori*. Implementing tools that pretend to extract information from this kind of data (for example, as in video summarization) has proven problematic. There is no systematic approach to doing so, nor a *de facto* solution.

In recent decades, many feature extraction techniques for multimedia content have been developed, but these techniques rely heavily on fixed rules, which may not be sufficient to capture the complexity of the data or may extract information irrelevant to the target task.

Deep learning (DL) can produce an abstract representation of the data that proves useful for performing tasks such as classification, summarization, transformation, etc., displaying great generalization capabilities. Such representations are obtained by passing the data through a sequential set of neural layers with different structures. These layers are responsible for extracting increasingly representative and meaningful features. As depicted in Figure 2.5, the data is first "flattened" into a vector, then goes through a set of layers to finally output a vector. Any of the output vectors from the layers resembles the input and may serve as a proxy to its modal content. The information embedded into this feature vector entirely depends on the input data, the task and the training of the DL network.

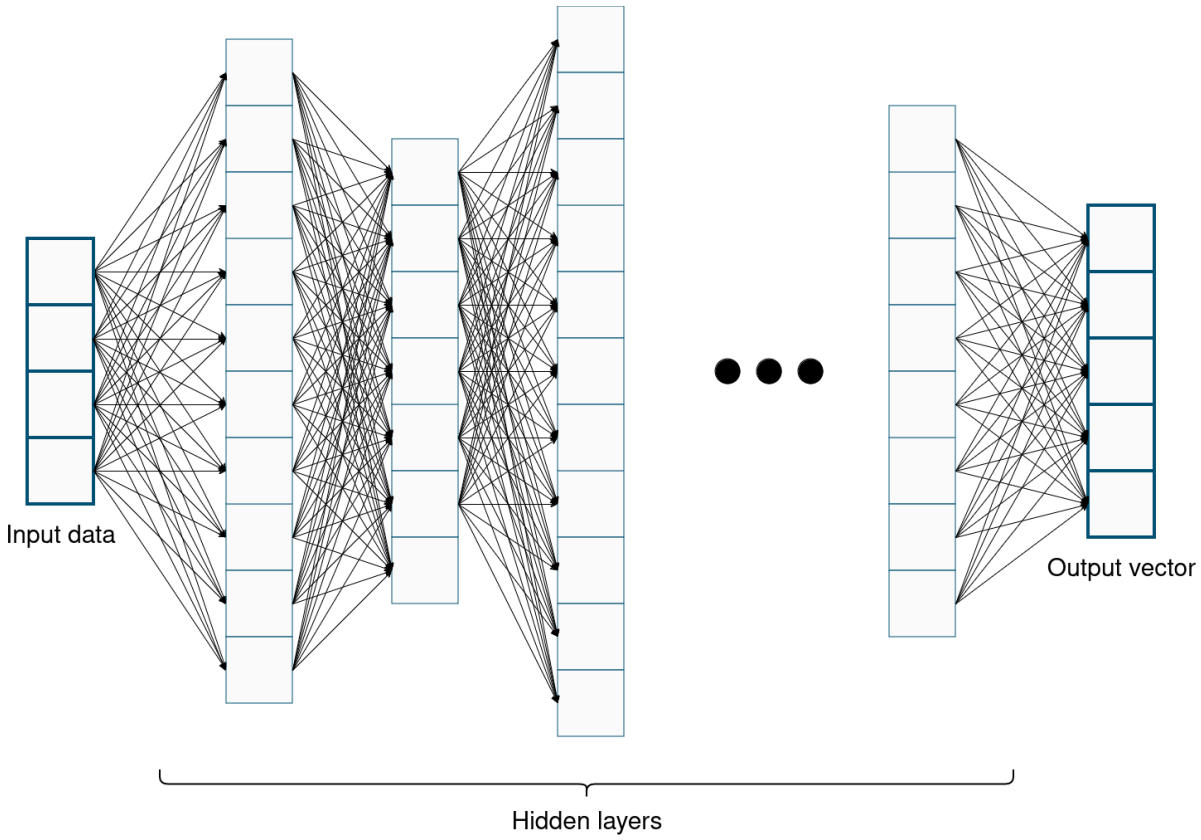

Figure 2.5: Schema of an artificial neural network.

The key to deep learning feature extraction is that its first layers tend to extract useful features in a broad set of tasks, while the final layers are keen on transforming those

features to perform the specific training task. Thus, initial layers of a deep neural network can be trained with a general and massive set of training data and then be reused in other networks or other tasks. This technique is known as *transfer learning*, and is one of the reasons behind the recent popularity of DL and the remarkable results it achieves.

Although multimedia data presents an unstructured nature, it usually contains some "patterns" that are useful. For example, in visual data, nearby pixels tend to be correlated, and in text data, words show cross-dependencies based on language grammar (adjectives depend on nouns, etc.). The architecture of a deep neural network ought to be *biased* towards some data modality to extract features more efficiently by being aware of the usual nature of the content. An example of this biased approach is convolutional neural networks (CNNs) and the convolutional layers, which perform convolution operations throughout to extract features from nearby locations within the layer-input vectors.

A biased DL architecture that has been successful when working with different modalities are the so-called transformers. The following subsection gives a brief introduction to these architectures, in a similar way as explained in [10].

2.3.1 Transformers

A transformer is a neural network with a sequence of elements (tokens) as input and a sequence of elements as output. Thus, a transformer maps sequences from one vector space to another. The idea behind this design is that the output vector space presents useful properties for a downstream task, like classification or translation, while the inputs are homogenized to ease the learning process.

The sequence transformation is determined by the transformer's attention. This mechanism generates the output tokens by processing the corresponding token from the input sequence in a way that is influenced or tuned attending to the rest of the input tokens and the corresponding token itself. Thus, a transformer can grasp the interrelations between the different elements of the input query.

The attention mechanism is suitable with textual data, since the meaning of each word or semantic structure (which in this case would be the tokens) is influenced by the rest. Nonetheless, transformers have demonstrated to be very effective with visual data as well, even outperforming CNN architectures in some tasks and applications. In the following pages, a deeper insight into transformers will be presented, as these structures may be used to connect the visual data with its semantic meaning.

As it was said, the input of a transformer is a set of tokens. A single token is represented as a vector $\mathbf{x}_n$ of fixed dimensionality, D. Thus, the input set of tokens for the basic layers of a transformer can be mathematically expressed as $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, and the output tokens as $\{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$. Similarly, these sets can be expressed as $\mathbf{X} = \left(\mathbf{x}_1^\top; \ldots; \mathbf{x}_N^\top\right)$ and $\mathbf{Y} = \left(\mathbf{y}_1^\top; \ldots; \mathbf{y}_N^\top\right)$ matrices of dimensionality $N \times D$.

Each output token is generated by a weighted sum of the input tokens, giving more importance (or attention) to some input tokens over others.

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm}\mathbf{x}_m, \text{ subject to: } a_{nm} \geq 0, \sum_{m=1}^{N} a_{nm} = 1 \tag{2.1}$$

A transformer architecture defines a_{nm} through the self-attention mechanism. Given the output token $\mathbf{y}_m$, the token $\mathbf{x}_m$ will be called **query**. All of the input tokens $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, including $\mathbf{x}_m$, will be called **keys**. The query will be compared against each key to obtain the coefficients using a comparison function. An example of these functions is the dot product:

$$a_{nm} = \frac{\exp\left(\mathbf{x}_n^\top \cdot \mathbf{x}_m\right)}{\sum_{m'=1}^{N} \exp\left(\mathbf{x}_n^\top \cdot \mathbf{x}_{m'}\right)} = \text{Softmax}\left(\mathbf{x}_n^\top \cdot \mathbf{x}_m\right) \tag{2.2}$$

It should be noted that in the Equation 2.2, the dot product is performed between the query ($\mathbf{x}_m$) and the corresponding key ($\mathbf{x}_n$). This operation controls the degree of importance the query $\mathbf{x}_m$ has in the generation of the output corresponding to the key $\mathbf{x}_n$.

Matricial notation can be used to write the whole layer operation in compact form:

$$\mathbf{Y} = \mathbf{A}\mathbf{X} = \text{Softmax}\left(\mathbf{X}\mathbf{X}^\top\right)\mathbf{X} \tag{2.3}$$

It should be noted that the transformer lacks any trainable parameters once the learning has been completed. Also, the comparison operation focuses equally on all the tokens, whereas a more "biased" comparison may be more beneficial. This problem is solved by transforming $\mathbf{X}$ through a trainable linear projection, $\mathbf{U}$:

$$\tilde{\mathbf{X}} = \mathbf{X}\mathbf{U} \tag{2.4}$$

Therefore:

$$\mathbf{Y} = \text{Softmax}\left(\mathbf{X}\mathbf{U}\mathbf{U}^\top\mathbf{X}^\top\right)\mathbf{X}\mathbf{U} \tag{2.5}$$

This change comes with a problem: the matrix $\mathbf{X}\mathbf{U}\mathbf{U}^\top\mathbf{X}^\top$ is symmetric, and symmetry is not a suitable property for this kind of processing, since if the matrix passed to the softmax function is symmetric, the attention between tokens becomes bidirectional. Bidirectional attention means that the importance given to a key token $\mathbf{x}_k$ when processing a query token $\mathbf{x}_q$ is the same as the importance given to the query token when it acts as the key of the previous key token. That bidirectionality may not be useful if the goal is to obtain a semantic representation of the input.

So, a separate key, query and value matrices are defined as the linear projections of input tokens:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(q)} \tag{2.6}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(k)} \tag{2.7}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}^{(v)} \tag{2.8}$$

Therefore:

$$\mathbf{Y} = \text{Softmax}\left(\mathbf{Q}\mathbf{K}^\top\right)\mathbf{V} \tag{2.9}$$

The softmax function as it is presents one last problem, related to the training process of neural networks. The gradient used in each training step becomes too small to reach convergence. To tackle this, a scaling is applied:

$$\mathbf{Y} = \text{Attention}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}\right) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{D_k}}\right)\mathbf{V} \tag{2.10}$$

with D_k the number of columns of $\mathbf{K}$ matrix. Figure 2.6 represents a schema of this layer.

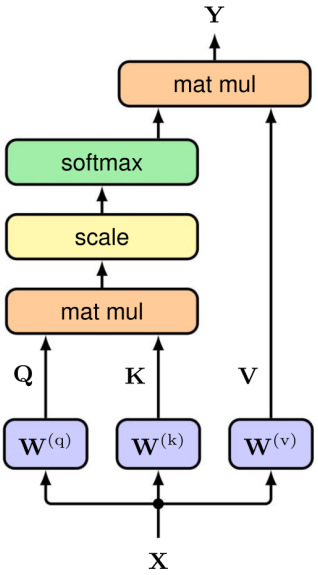

Figure 2.6: Schema of an attention layer. Extracted from [10].

The previously described calculations are the core of a transformer architecture. Several supplementary operations can be included starting from this basic layer to improve training or achieve stronger generalization capabilities. Examples of these are residual connections, positional encoding, or applying several connection heads in parallel [10]. In all cases, the attention mechanism is the fundamental core behind the great potential of transformers.

2.3.2 Multi-modal transformers

Although transformers have been considered the NLP-biased models, the attention mechanism they implement is prone to a good generalization among other modalities of data, such as image, audio, or speech. Thus, a transformer can be used as long as a proper tokenization and detokenization method is defined; that is, a way of translating the content into a sequence of vectors and vice versa.

An example of a transformer application to visual data is the work by Dosovitskiy *et al.* [11], where the original transformer architecture is trained on well-known vision datasets to perform classification tasks. Despite the fact that transformers had not been specifically developed to work with visual contents, like CNNs, this model yielded impressive performance compared to traditional convolutional networks like ResNet [11]. Following this lead, transformer architecture has also been applied to video data more recently [12].

Having impressive performance with models that are not biased towards a single multimedia format, or modality, opens the door for combining several modalities of data into one model to perform more complex tasks. In this context, CLIP appeared in 2021 to combine visual and textual data using transformer architectures [13]. The approach proposed by CLIP is to learn perceptual information present in images from natural language supervision rather than from specific classification labels.

One thing that should be noted is that, in essence, CLIP is not a model. Rather, it is a training method that aims to tune a vision model and a natural language model coherently with each other, so the common embedding space retains semantic information, that is, the text and images can be related on this abstract space. The trained architecture in the original paper introducing CLIP combines the use of a vision transformer (which in most implementations is Dosovitskiy's ViT) and a NLP transformer, which are trained jointly to generate a multi-modal embedding space. The pre-training task starts from a series of N pairs that include an image and an associated text: $\{image_i, text_i\}_{i=0}^{N}$. Each of these images are introduced into the vision transformer, and the same process is applied to texts with the text transformer. As a result, we obtain a set of representations in the form of embedding tuples[2], $\{I_i, T_i\}_{i=0}^{N}$. Then, each image-text pair is compared through a cosine similarity function:

$$\text{cosine similarity}(I_i, T_j) = \frac{I_i \cdot T_j}{||I_i|| \cdot ||T_j||}; \; i = 0, \ldots, N; \; j = 0, \ldots, N$$

The result is a $N \times N$ matrix that contains all the similarities. The target of the pre-training process is to maximize similarity when $i = j$ (the matrix's diagonal, when the text corresponds to the image) and minimize it otherwise (for the unmatched pairs). This training technique is known in the literature as contrastive learning.

Figure 2.7 presents the CLIP architecture in the encoders pre-training stage.

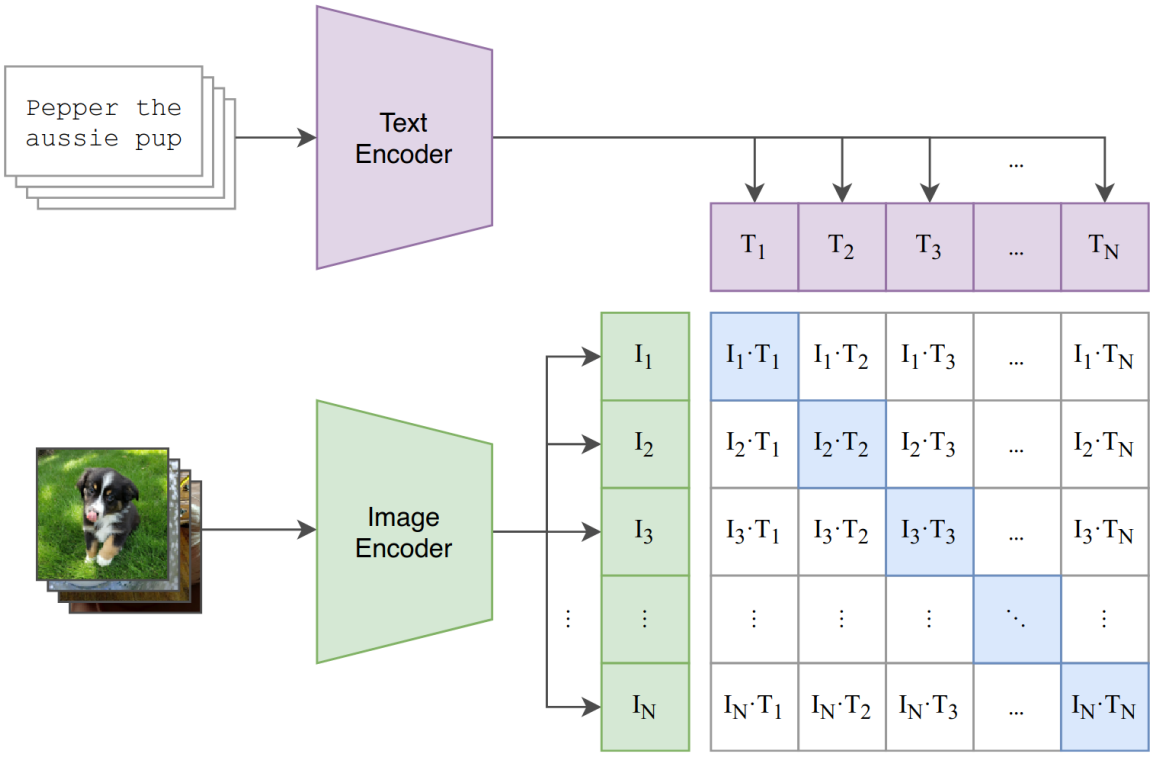

Figure 2.7: CLIP architecture for contrastive pre-training. The diagonal of the comparison matrix is maximized through training while the rest of the elements are minimized. Extracted from [13].

[2]An exception to the common notation of representing vectors with lower-case bold letters is taken here in order to follow CLIP's notation. Nevertheless, both I_i and T_i are vectors.

The success of CLIP lays with its vector space where text and image embeddings can be compared and related: the embedding of a cat image is prone to yield a high cosine similarity with texts like "a picture of a cat," "a cat standing up," "a furry animal with four legs," etc. CLIP's paper proposes the use of the architecture for image classification, by measuring the similarity of the image vector with the vector of a prompt like "a picture of a {*class name*}." Figure 2.8 represents this use of CLIP for classification purposes.

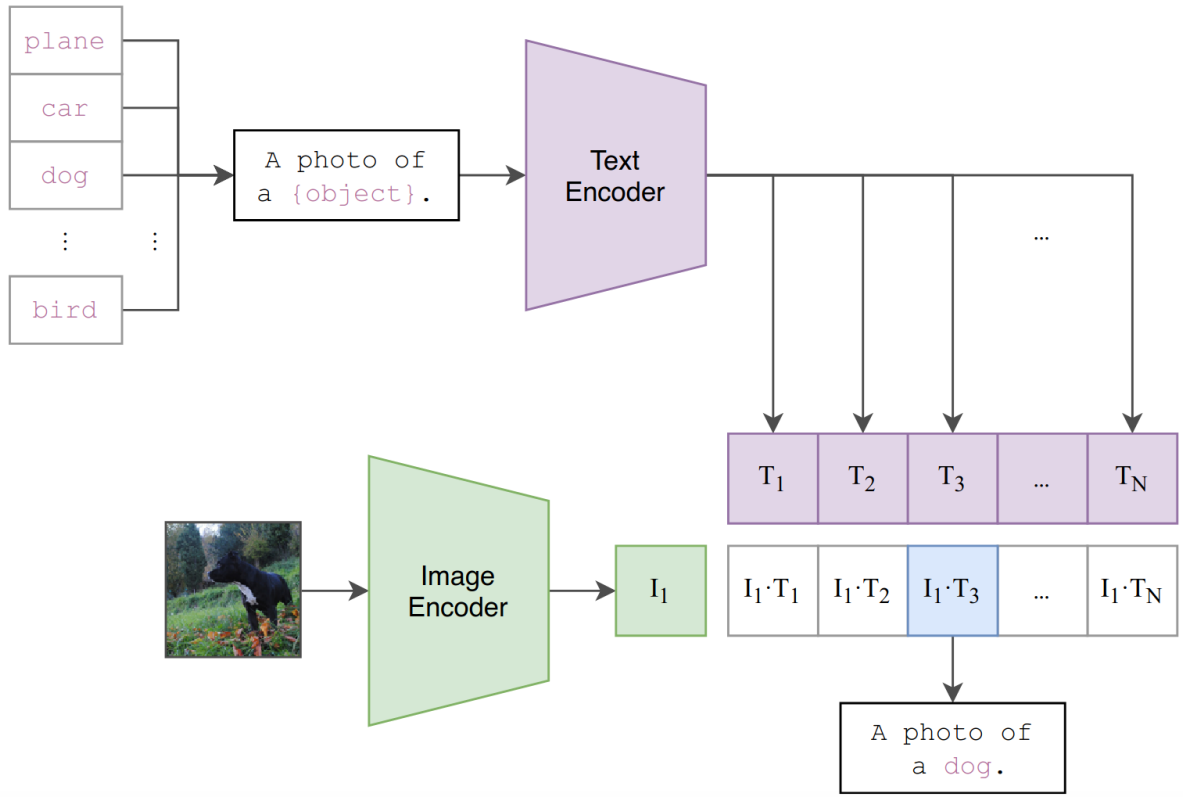

Figure 2.8: Image classification approach presented in CLIP. Extracted from [13].

2.3.3 Applications of CLIP

To end this section, a list of applications of the CLIP architecture will be presented.

- Image classification: the work which presented CLIP architecture explores this task, by comparing the embedding of images to the embedding of a text such as "This is a photo of {*category*}" [13].
- Semantic image retrieval: The embedding of an image generated through the visual transformer of CLIP contains the semantic meaning of the image. This fact can be used to implement search systems which look for elements semantically similar to some input, whether that input is a text or an image as well, comparing the prompt embedding to the elements' embeddings. An example of this application implementation can be found at [14].
- Content moderation: websites and social networks that allows users to post images and visual content must implement a system which automatically removes any

non-acceptable uploads, such as erotic, violent or offensive images. This moderation can be implemented with CLIP by classifying images into suitable or unsuitable content. The great zero-shot capabilities of CLIP makes it a good choice for this kind of task [15].

- Image captioning: the authors of [16] have combined the use of CLIP visual transformer with GPT transformer to generate captions (short descriptions) of an image automatically. This has applications in metadata generation and in accessibility.

2.4 Vector databases

The vector space generated by artificial intelligence models allows the comparison of unstructured elements within a trained space defining a semantic domain, just by computing similarity between embeddings. Common applications of this technique include reverse image search [17] and recommendation systems [18], among others. Furthermore, architectures such as CLIP have introduced the possibility of confronting similarities across modalities, like a text and an image, in their tokenized form. In such case the logic for operating in the vector space is the same.

In current industry applications, projects and systems must cope daily with enormous amounts of unstructured data, such as multimedia assets. This represents a significant challenge when trying to build efficient and valuable applications. Vector representations unveil a world of possibilities for processing these assets, but they also present a challenge when managing large amounts of vector data. For example, for a 10-minute video, assuming that we extract 25 frames per second and that the ViT encoder in the original CLIP implementation is used, the total number of vectors reaches 15 K, making 7.7 M coefficients in total.

In this context, Vector Database Management Systems (VDBMS) provide tools to store and manage high-dimensional vector data. These tools include an implementation of a data model specifically designed to store high-dimensional vector data, instead of a generic model that aims to be compatible with a wide range of data types (strings, integers, dates, documents, etc.), as well as the indexing algorithms specifically designed for a rapid retrieval and search of assets from their vector representations stored in the database [19].

Figures 2.9 and 2.10 represent the usual data flow for these databases. In Figure 2.9 we represent the ingestion pipeline. All the unstructured data (or a relevant sample of it) undergoes a vectorization stage, to compute the embedding vectors. The vectorization operation embeddeds information into fixed-length sequences (i.e., the vectors). It is usually implemented through a deep learning model like a transformer or a CNN. Then, the vectors are inserted into the database, along with some metadata related to the original unstructured elements (e.g., where the new item is allocated, what it contains, etc.). The database automatically creates an indexing structure to allow quick and reliable searches –providing the means to be queried–, to filter the data, and to allow results fetching.

In Figure 2.10, the query pipeline is represented, starting from the generation of prompt data, typically a search image or a search text. This data undergoes exactly the same vectorization as the one used in the ingestion stage, obtaining a query vector. The query vector is sent to the vector database, and the database uses its index structure to search

for the most similar vectors (according to some predefined similarity function). Finally, the result returned to the application is some form of the metadata associated with the unstructured element corresponding to the filtered vectors.

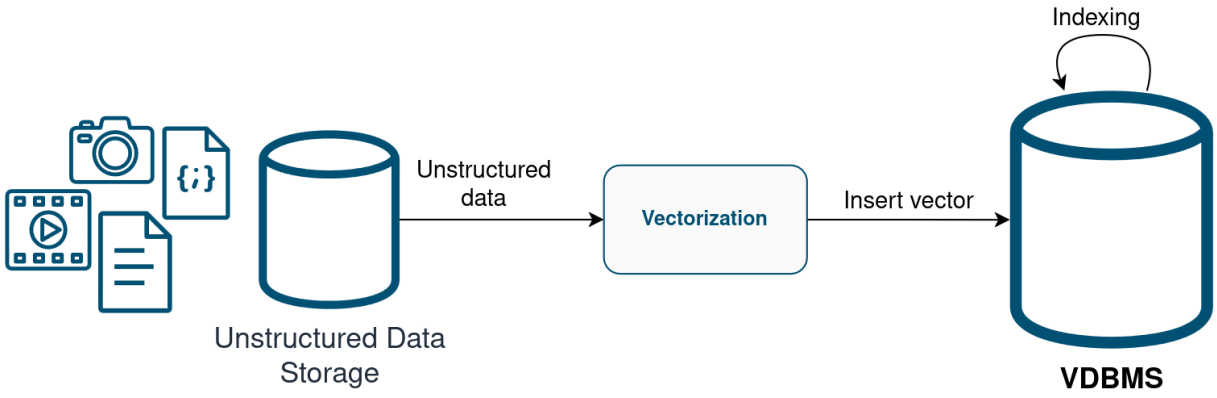

Figure 2.9: Information flow for ingestion in a VDBMS. Extracted from [19].

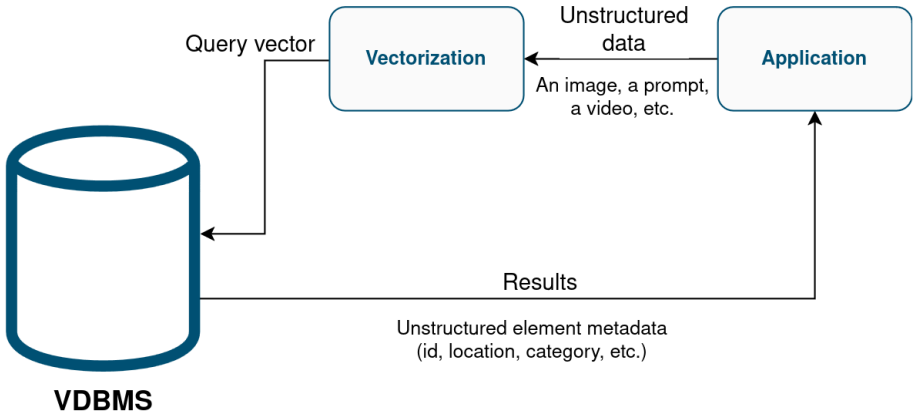

Figure 2.10: Information flow for querying in a VDBMS. Extracted from [19].

Mathematically, a search in a vector database can be expressed as follows. Given a database of $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ vectors of dimensionality D and a query vector $\mathbf{u}$ of dimensionality D, the search is the performance of the operation described in Equation 2.11 [20]:

$$i = \arg\max_i \{\text{similarity}\,(\mathbf{u}, \mathbf{x}_i)\} \tag{2.11}$$

The use of a query vector to retrieve the results makes this operation also known as querying, and needs to be computed *online* on the provided query.

2.4.1 Indexes

An index is a structure that organizes data *offline* so the *online* operation in Equation 2.11 is performed in a fast and accurate way. Ultimately, querying a vector database means computing a similarity function between the query and the stored vectors; the result is the vector that scored the highest similarity[3]. A vector database index arranges

[3]Actually, VDBMSs return not only the most similar vector but a ranking of the top-k most similar vectors, with k a parameter introduced by the user in the query operation.

vector data to reduce the number of similarity computations to solve the query without compromising accuracy. This is achieved by dismissing those that most likely will not produce the top-ranking results.

The general approach to these indexes is to divide the vector space into regions, reducing the search to one or a reduced set of operations [21]. The literature describes three main types of indexes: quantization-based, graph-based, and tree-based indexes.

2.4.1.1 Quantization-based indexes

Also known as Inverted File (IVF) indexes. Generally, the computation of such index first requires a k-means clustering on the vector space. Each database vector is assigned to the cluster whose centroid is closer to the vector in the form of an inverted list. Being $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ the set of vectors to cluster, and $\mathcal{C} = \{C_1, \ldots, C_K\}$ the set of clusters, these are obtained by the expression 2.12.

$$\mathcal{C} = \underset{\mathcal{C}=\{C_1,\ldots,C_K\}}{\arg\min} \left\{ \sum_{C_j \in \mathcal{C}} \sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i \cdot \boldsymbol{\mu}_j \right\} \tag{2.12}$$

with $\boldsymbol{\mu}_j$ being the centroid vector of the j-th cluster [22].

Then, when the database is queried, a set of $P \leq K$ clusters are explored, comparing the query vector to all the vectors contained in the P inverted lists. These P clusters are the ones with the closest centroid to the query.

Inside the inverted lists, the vectors can be encoded into a compressed format, through techniques like scalar quantization (affecting the elements of the vector into lower-bits integers) or product quantization (slicing the vectors in several pieces and quantifying each piece through a codebook) to reduce the storage space [20].

2.4.1.2 Graph-based indexes

The general idea of graph-based indexes is to build a graph whose nodes are the vectors of the database. The search process is implemented as a navigation through this graph following the path to the closest vector. Among the existing types of indices, HNSW is the most spread one.

HNSW builds its graph from the vectors as nodes, separating the links according to their length into different layers. The search is performed across the multi-layer graph. It starts in the layers with the longer-distance links until a local minimum is reached. Then, the algorithm moves onto the immediate lower layer to search for a local minimum [23]. The process repeats until a final solution is identified. We illustrate this in Figure 2.11. The first search is computed on Layer=2, where the nodes with the most prominent links sit. It moves onto Layer=1, finding a local minimum. The process repeats in Layer=0 until the final node is reached, in green.

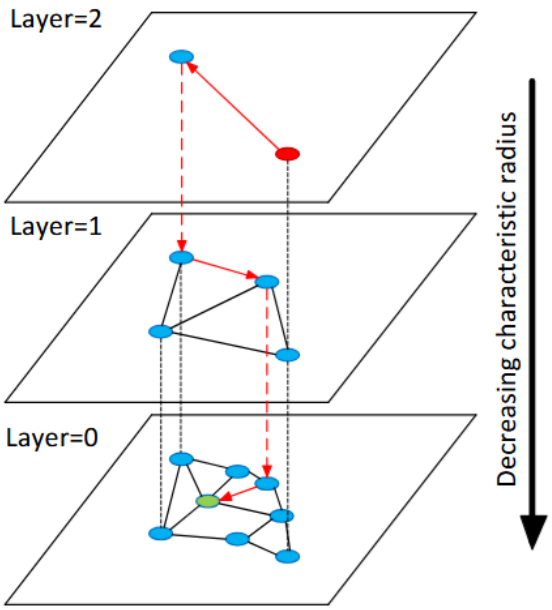

Figure 2.11: Illustration of the concept behind HNSW. The red arrows indicate the search path from the entry point (red dot) to the exit point (green dot). Extracted from [23].

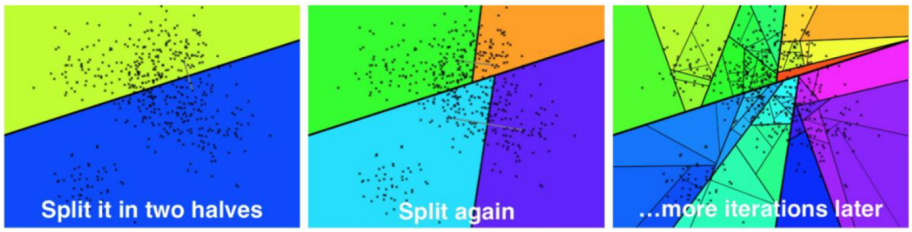

Figure 2.12: Visualization of the splitting process implemented in ANNOY. Extracted from [25].

2.4.1.3 Tree-based indexes

Tree indexes use a special case of graph, since a tree is a special type of graph. One widely-spread implementation of tree-indexing is ANNOY, developed at Spotify [24]. ANNOY creates a tree by dividing the vector space into two halves recursively, until a stop condition is reached: a split having less then K elements.

The splitting process works as follows: first, two random vectors are chosen, and a hyperplane is built equidistant from the two vectors, creating two hyperspaces represented each by a child node. For each child node, the splitting process is repeated, creating two new child nodes per prior child nodes. This process keeps repeating until the stop condition is reached, creating a set of leaf nodes with less than K vectors. The process is illustrated in Figure 2.12. Then, the querying process consists on traversing the tree by determining on which side of the different hyperplanes the query vector lies, until a leaf subspace is reached [25].

2.4.1.4 Indexes comparison

The current state of the art demonstrates that IVF and graph-based indexes like HNSW are the *de-facto* indexing standards for the industry, being tree-based indexes less spread due to their worse performance, especially with high-dimensional data [26].

Comparing IVF and HNSW, several magnitudes must be taken into account: **recall**, **search time**, **memory usage** and **construction time**.

The Faiss index library offers a decision tree to choose between indexes [20]. Their main criteria for choosing an IVF index over a HNSW index is the size of the dataset stored in the database: if the dataset is too large, the memory usage and construction time of HNSW turns unfeasible for real-world scenarios, so IVF indexes become a better option. The number of vectors in the dataset proposed as a threshold is 10 million; if this threshold is surpassed, IVF index is the best option; if not, HNSW is the suitable choice. In this work, when we had to choose an index, this was the followed criteria.

Despite these trade-offs, graph-based indexes always show the best performance in terms of recall and search time [26], and HNSW seems to be by far the dominant indexing algorithm in vector databases environment [27].

2.4.2 Vector database implementations

According to several market studies, vector databases have enjoyed continuous growth for years [28]. Specifically, the vector database market is expected to grow at a CAGR of 21.9%, from a value of USD 2.2 billions in 2024 to USD 15.1 billion in 2034 [29]. The main driver for this technology growth is the seamlessly integration of vector databases and artificial intelligence frameworks and applications. These are empowering applications such as recommendation engines (used by companies like Netflix, Amazon or Spotify) and search engines (used by Google or Bing) [30], among several others.

Figure 2.13 presents the technologies used by companies for vector storage. The graph evidences that the most used solution is MongoDB vector management capabilities, followed by Redis. These two technologies are not vector-specific solutions (Redis is not even a persistence-focused database), so it is clear that companies are not massively adopting VDBMS in their applications, and still rely on vector management extensions of general purpose databases. This fact is expected to change over the following years due to the need for fast and efficient vector management.

The review in [21] compares several popular vector-specific databases by studying the implementation of a set of features. A summary of the results of their analysis is included in Table 2.1. The Table covers five widely-spread open-source VDBMSs.

All of these are fairly new; the oldest one was released in 2019, and all of them rely on the Python programming language in combination with other languages. The implementation of distance metrics shows the dominance of the cosine distance (inner product for CLIP is an equivalent metric); the most implemented index is HNSW. Finally, it is also clear that all of them are compatible with distributed deployments, which helps to manage the workload of the application.

Figure 2.14 shows another comparison between different VDBMSs. The features are

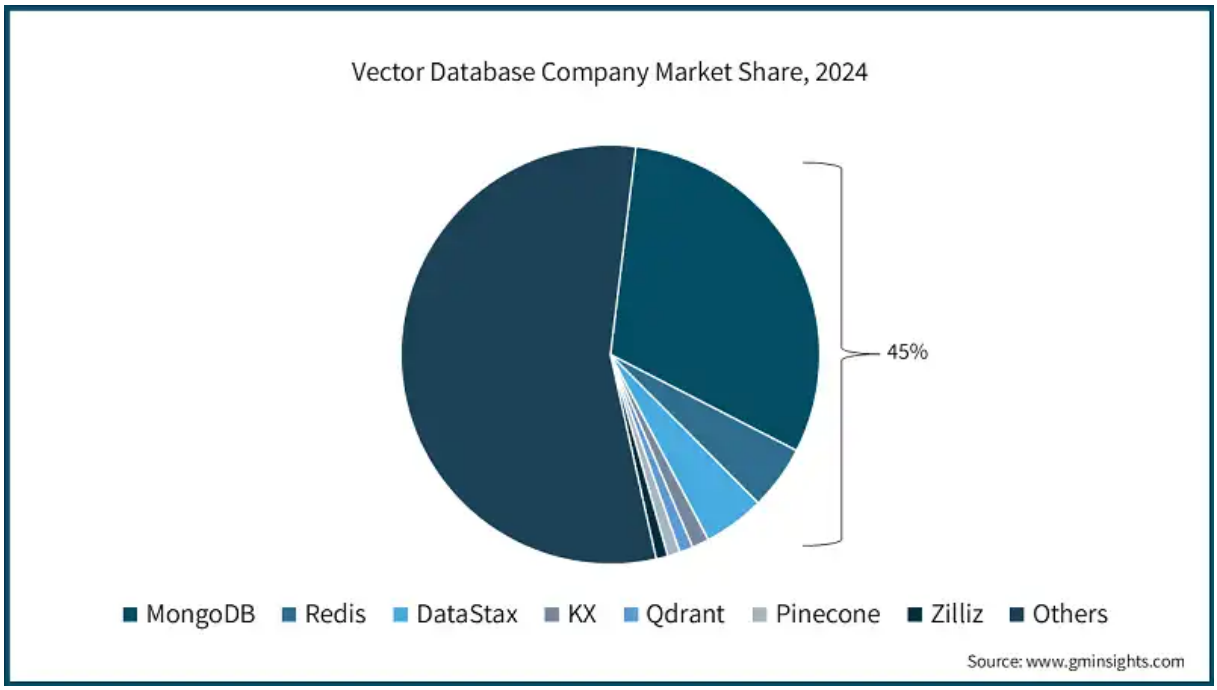

Figure 2.13: Division of the market share according to technology. Extracted from [29].

Table 2.1: Comparison of different vector databases. Extracted from [21].

	Chroma	**Weaviate**	**Qdrant**	**Vespa**	**Milvus**
Release Date	Oct. 23, 2022	May 21, 2019	April 6, 2021	Sept. 22, 2021	Oct. 21, 2019
Open Source	Yes	Yes	Yes	Yes	Yes
Metrics	Cosine	Cosine	Inner Product, Cosine, Euclidean	Inner Product, Euclidean, Angular, Geo Degrees, Hamming	L2, Inner Product, Hamming, Jaccard, Tanimoto, Superstructure, Substructure
Language	Python/ TypeScript	Python, Go	Python, Rust	Java, Python	C++, Python
Distributed	Yes	Yes	Yes	Yes	Yes
Indexing	HNSW	HNSW (PQ), customized HNSW	Customized HNSW	HNSW, BM25	HNSW, Annoy, IVF_PQ, IVF_SQ8

placed in a Cartesian plane. The vertical axis represents whether the system is hosted on a local machine or have to be deployed in a cloud. The horizontal axis represents whether each database relies on a client-server architecture or is deployed without a server. The dominance of the client-server architecture is clear, while many databases offer both on-premise and cloud deployments.

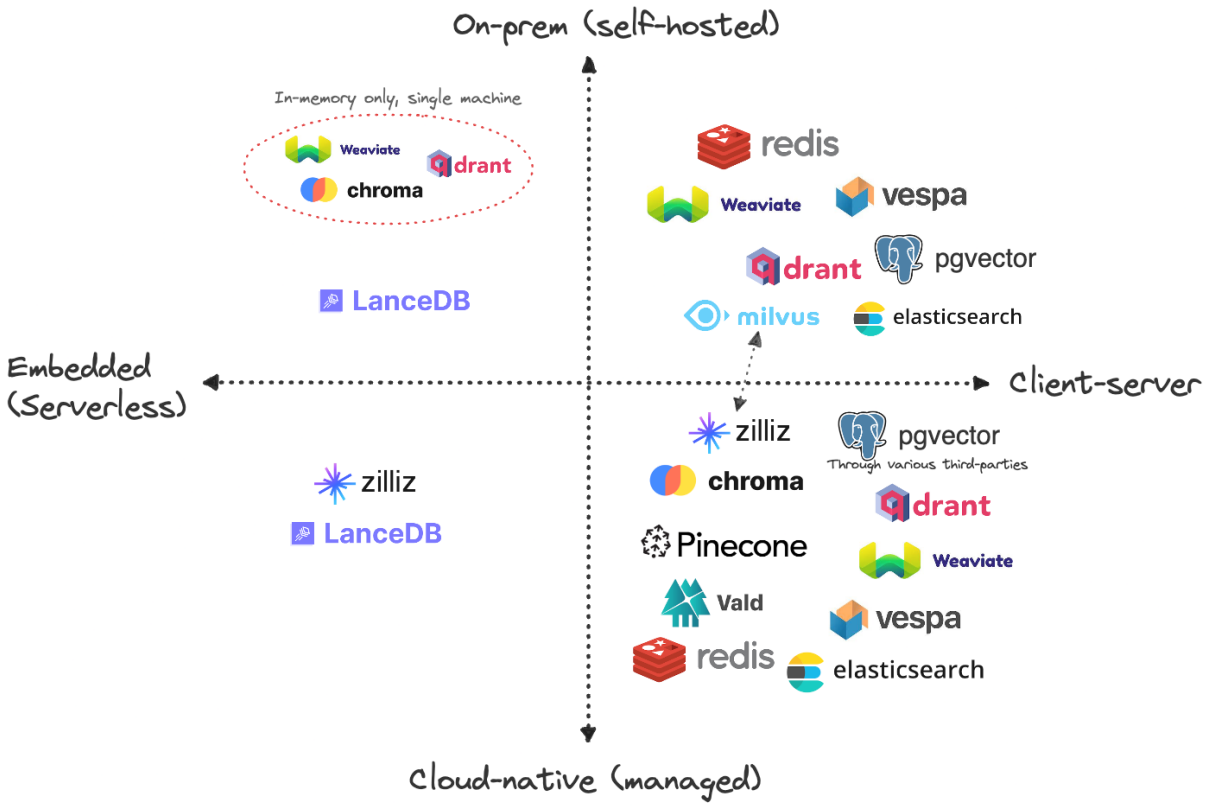

Figure 2.14: Comparison graph for several vector databases. Extracted from [27].

2.4.2.1 Traditional SQL for vector storage

Aside from VDBMSs, which are specifically developed to manage vector data, many projects use traditional SQL database PostgreSQL, as a vector database. By leveraging its open architecture, one can develop extensions that can store and work with vector data [31] [32]. These extensions basically define a vector data type and implement indexes interfaces (IVF and HNSW). The PostgreSQL engine can perform the operations any other vector database would.

The work in [33] performed a comparative analysis between the Faiss library [20], which is a well-known vector indexing tool, and PASE [32], a PostgreSQL extension for high-dimensional vector data.

The main outcome of this work is that the relational model is not a fundamental limitation for the performance of a vector database. Although it was demonstrated that Faiss library outperformed PASE in every aspect, the gap between them was not because of PASE executing on top of a SQL database, but because PostgreSQL engine had not optimized for vector indexing operations in terms of memory and computation parallelism.

This means that, eventually, **a SQL-compatible vector database could** meet market expectations and **be a solution** widely spread in the industry **for vector management**, due to the combination of the relational scheme and the ability to work with high-dimensional embeddings.

In this context, PostgreSQL is a promising application, due to its great extensibility properties and its huge penetration in the data-oriented market [34]. If a company that has

its multimedia data management framework based on PostgreSQL and wants to start using embedding representations, it could well implement a vector management tool embedded inside its already existing PostgreSQL infrastructure. That being said, a comparison with popular vector database is worth the effort.

2.5 Video surveillance datasets

Every DL-based application needs some dataset to test its abilities. We used a video surveillance dataset known as UCF-Crime.

2.5.1 UCF-Crime dataset

UCF-Crime dataset (from this point forward, UCF dataset) was originally gathered for tasks related to anomaly detection and classification. It includes 1900 untrimmed videos of CCTV cameras, totalling 128 hours of footage. From these, 950 are tagged as normal videos, and 950 are anomalous, each of these last being classified into one of 13 categories: *abuse*, *arrest*, *arson*, *assault*, *road accident*, *burglary*, *explosion*, *fighting*, *robbery*, *shooting*, *stealing*, *shoplifting* and *vandalism*. This material was downloaded from the web by introducing different text prompts into online video-sharing platforms (Youtube and LiveLeak) and filtering the unsuited content afterwards. Regarding the terminology for Section 2.2, we will assume that each video file represents a single shot s of a scene S.

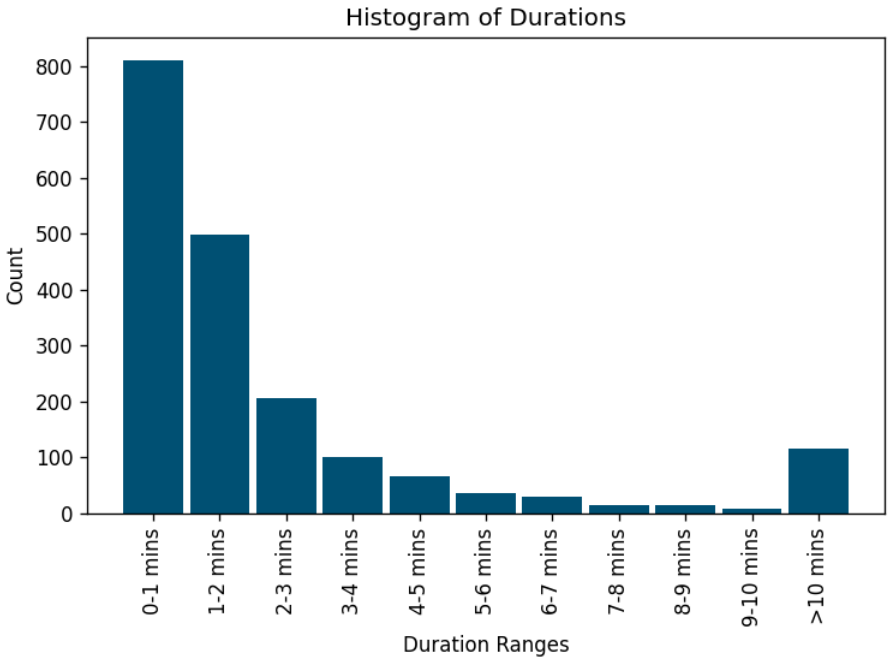

Figure 2.15: Bar graph representing the number of videos matching the duration in the predefined bins.

Figure 2.15 depicts the histogram of video durations, the number of videos that fall under each defined range. The UCF dataset mainly contains short videos, with longer videos rarer as the duration increases. Having a lot of short shots of different scenes

brings much diversity to the dataset, so the accuracy of the data retrieval can be better tested. However, it is also true that it is not a realistic dataset for a CCTV application, which is supposed to have very long videos of a few scenes. In CCTV scenarios, there is usually little relevant visual information, which is very short compared to the duration of the footage. Nevertheless, the indexed videos also include segments where no relevant information is observed, even if this is short.

Figure 2.16 shows the number of videos per anomalous category, leaving normal videos out of the analysis. For most of the categories, the dataset contains 50 videos; for *burglary* and *stealing*, it contains 100 videos; and for *road accidents* and *robbery*, it contains 150 videos. A greater amount of videos for one category means more diversity of scenarios but does not necessarily mean more footage since videos can be shorter.

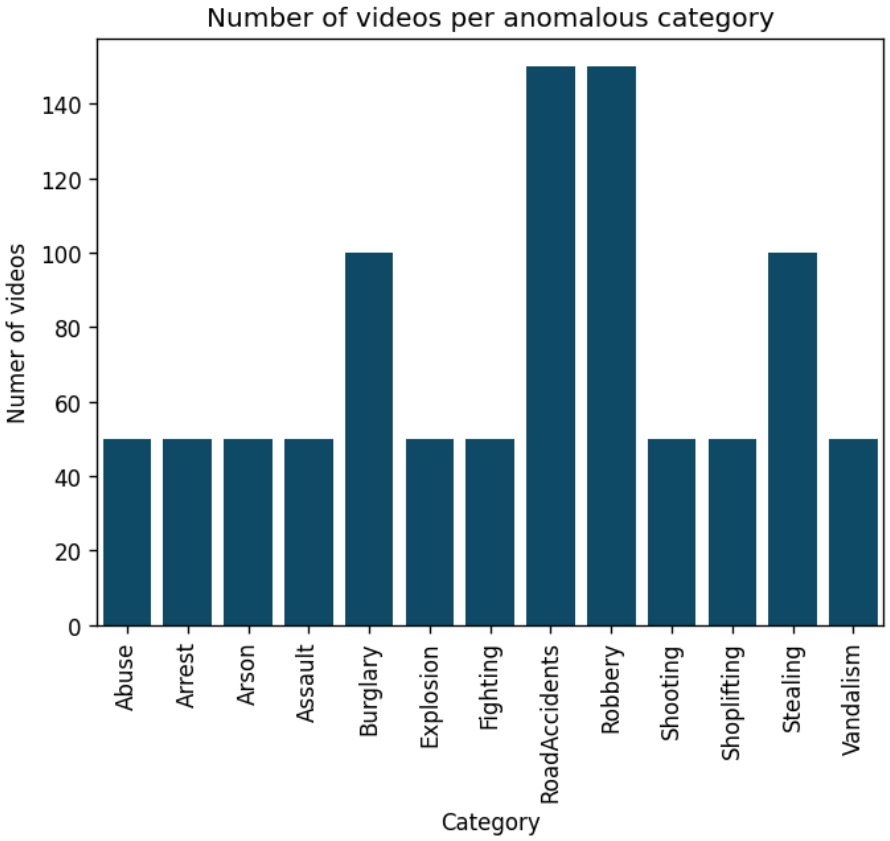

Figure 2.16: Bar graph representing the number of videos of each anomalous category.

Figure 2.17 shows the number of frames per anomalous category, and Figure 2.18 shows the percentage of frames belonging to each anomalous category. For this analysis, we assumed that all frames in a video tagged in an anomalous category should be anomalous. However, previous analysis of key frames and unsupervised clustering on CLIP evidenced that this might not be the case [35]. Section 3.3.2 will explain that, for each dataset frame, a vector representation was computed and stored. Thus, it is more revealing for this analysis to study the number of frames rather than the duration of the videos. Nonetheless, the frame rate of most of the videos is 30, presenting only 10 not anomalous videos with a frame rate of 25. Thus, the distribution presented by the number of frames is the same as the distribution of the seconds of footage per category. It can be seen that each of the categories has a close number of frames. Therefore, the visual data of the dataset presents

a great degree of diversity.

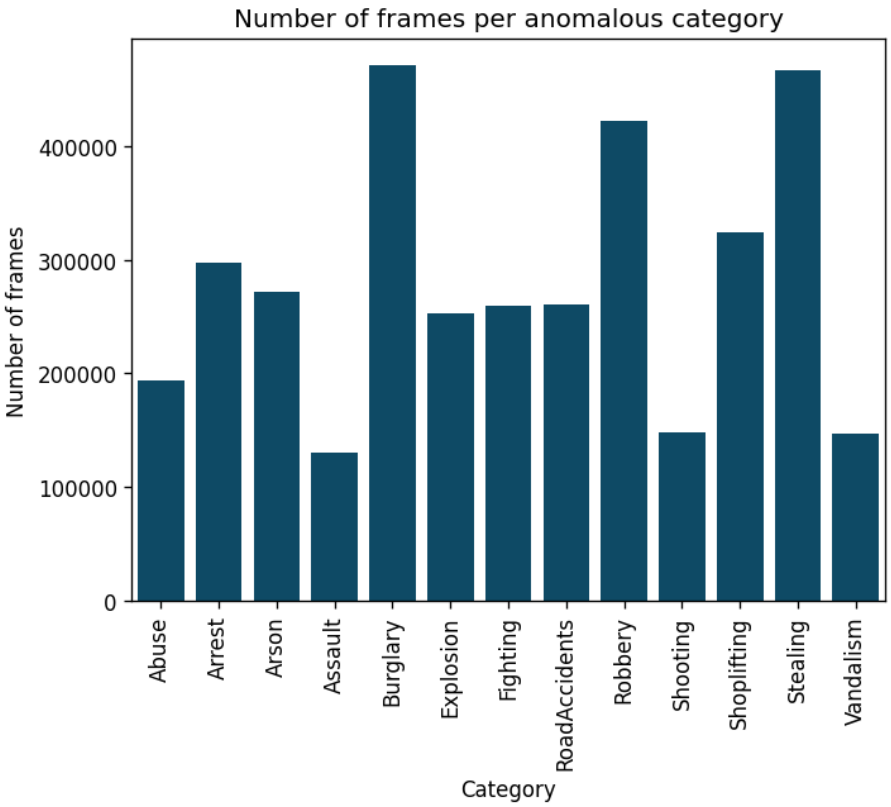

Figure 2.17: Bar graph representing the number of frames in each anomalous category.

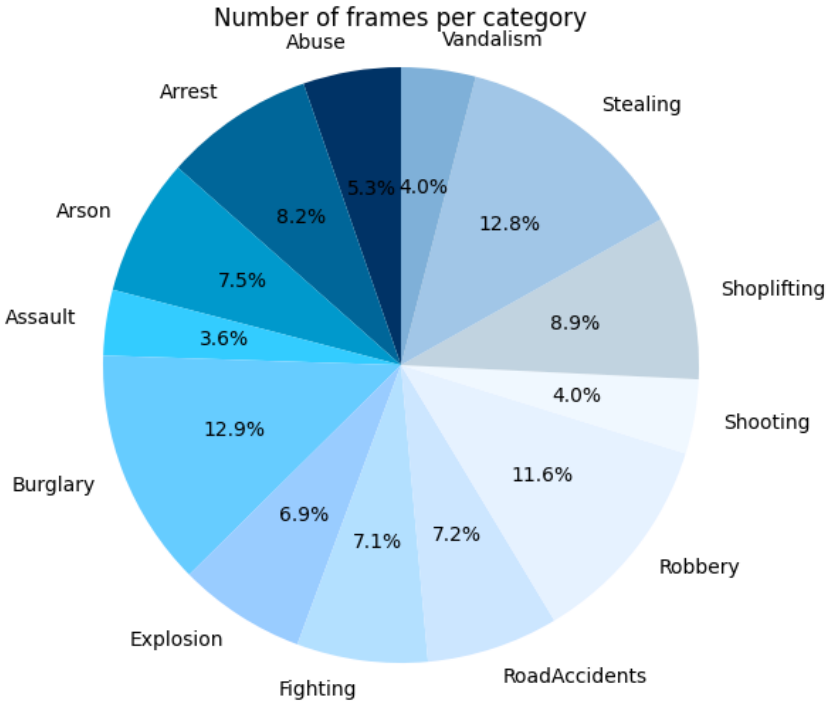

Figure 2.18: Pie chart representing the percentage of frames in each anomalous category.

In summary,UCF contains 950 anomalous videos and 950 normal (non-anomalous)

videos, several of which are quite long. The study conducted in this Master's Thesis showed that around 70% of the frames fall into the normal categories. This unbalance does not present a problem since real CCTV systems record mostly uninteresting data, so a video surveillance retrieval system must cope with the abundance of useless footage –i.e., lacking meaningful visual information to be retrieved.

Figure 2.19 presents a box diagram per each category, including normal videos.

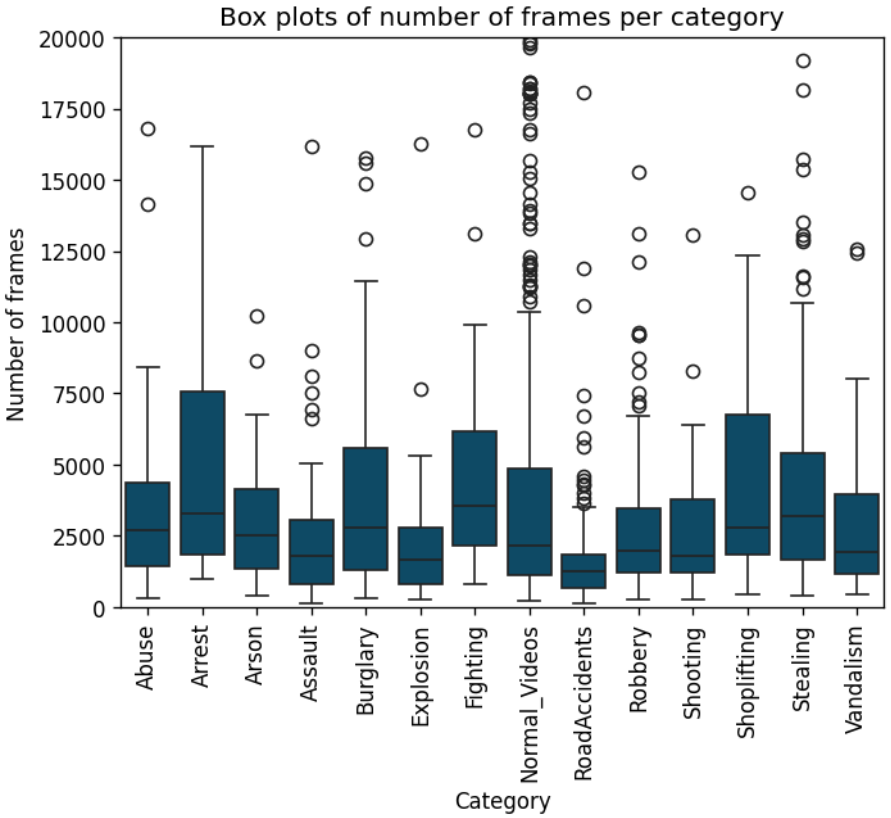

Figure 2.19: distribution of the number of frames per category.

2.5.2 UCA dataset

UCF dataset became very popular in the Computer Vision field, especially in tasks related to anomaly detection [36]. In 2023, UCF-Crime Annotation dataset (UCA) was created to provide UCF dataset with natural language descriptions of the different events taking place in the videos.

Each annotation is made of a description of the event, the start second of the event and the end second. Thus, UCA dataset is just an event-segmentation of UCF dataset. The methodology for building this dataset was the following: first, a filtering of UCF dataset was performed by discarding the videos unsuitable for natural language description (with occlusions or with low resolution), leaving a total of 1854 videos. Second, human annotators provided a detailed description of each event in the video (along with the start and end times). This resulted in a total of 110.7 hours of annotated videos.

The descriptions provided by UCA are of no interest in this work. The only feature of

UCA we used is the fact that it divides UCF in a set of short videos, which can be used as summaries.

Table 2.2 shows a comparison between UCF and UCA datasets. Certainly, UCA does not contain annotations for every UCF video.

Table 2.2: Comparison between UCF-Crime dataset and UCA dataset.

	UCF	**UCA**
Number of videos	1900	1854
Categories	14	14
Hours of video	127.51	110.66
Number of frames	13,768,423	11,939,961[4]
Number of annotations	-	23,542

[4]The actual number of unique frames indexed is less than this number, since this number was calculated by adding the number of frames of each annotation, and annotations may present overlaps between them.

Chapter 3

Implementation

This Master's Thesis developed a video surveillance retrieval application that uses multi-modal embeddings to generate video summaries of the surveillance footage. The goal is to develop the application as a set of components that can cope with vectorized representations of multimedia data and see how those components could engage with systems present in today's technological environment, focusing in this case on video surveillance.

This chapter describes the implementations conducted, giving an architectural overview, followed by a detailed specification of each of the system components and their layers. The Master Thesis does not address the implementation of a data ingestion pipeline for real-time streaming cameras. Instead, it uses a video surveillance dataset and describes its processing *offline*. The storage and organization of data were entirely covered to focus on retrieval. The following sections focus on the implementations completed.

3.1 Introduction

A video surveillance system, as understood in this Master's Thesis, covers two main pipelines: data ingestion and data retrieval. Data ingestion is a pipeline that is responsible for continuously capturing the images from the video endpoints and storing them in a video database. Data retrieval is a pipeline that searches (*online*) for specific video content obtained that was already indexed and stored (*offline*) as part of the data ingestion pipeline. This is usually carried out following some search criteria (source camera, date, hour, event taking place, etc.).

3.1.1 Data ingestion

We ingested the UCF-Crime [37] into our databases using the video files as the input data. We considered four approaches for video ingestion, which can be seen in Figures 3.1 to 3.4. These four approaches are based on Cisco's Video Surveillance solution [7], described in Chapter 2 and illustrated in Figure 2.4.

The first approach, represented in Figure 3.1, assumes an Encoding Server that sets a streaming session with the Media Server proxy and encodes the frames to embeddings

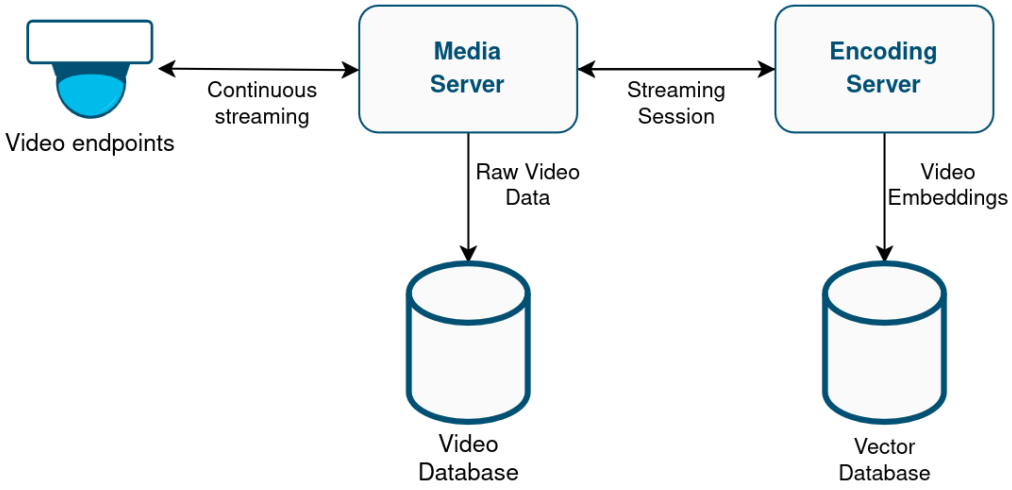

Figure 3.1: An external embedding server that acts as a client endpoint is responsible for encoding videos into embeddings

as they come. The advantages of this approach are that it does not change the logic of Cisco's solution and separates the video plane from the embedding plane, making it an affordable migration solution for legacy video surveillance systems. The disadvantage is that greater network bandwidth is required since the encoder needs a streaming session that is permanently active.

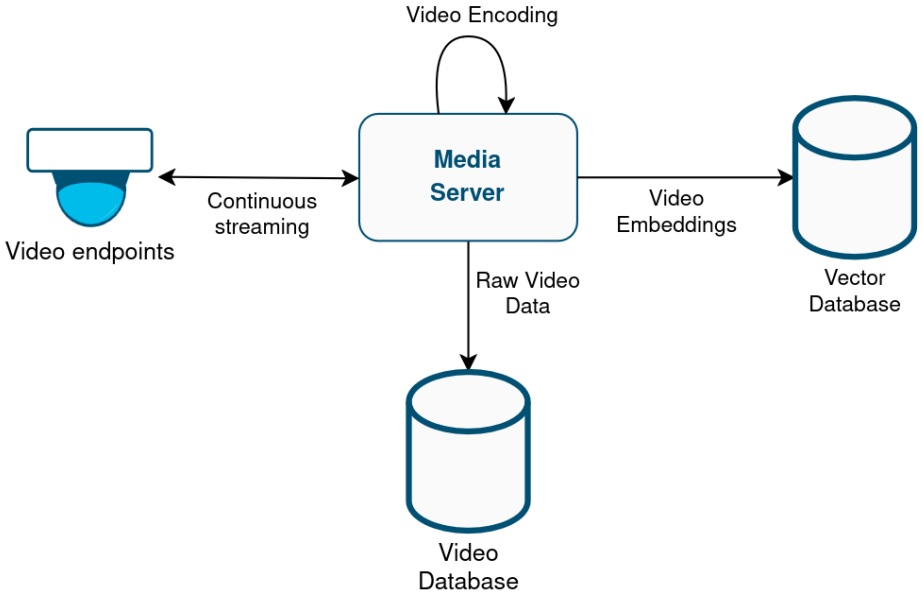

Figure 3.2: Media servers are responsible for encoding videos into embeddings.

The second approach, represented in Figure 3.2, assumes the embedding operation is performed in the Media Server, and then the last stores the embeddings in the vector

database. The advantage of this approach is that all the video processing and storage operations are isolated into one single component, which hides much of the complexity of the systems and provides greater data consistency. The disadvantage is that, for legacy systems, Media Servers must be updated and scaled since the embedding operation requires a lot of computing power.

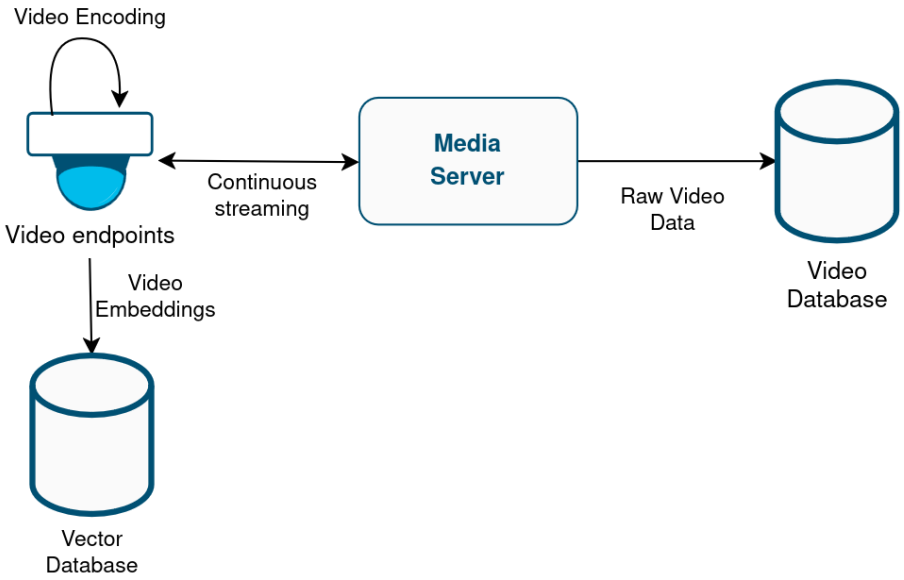

Figure 3.3: Video endpoints are responsible for encoding videos into embeddings.

The third approach, represented in Figure 3.3, assumes the embedding operation is performed in the Video Endpoint itself, and then the last stores the embeddings in the vector database. A slightly different approach could be the Video Endpoint generating the embeddings and sending them to the Media Server along with the video data. Then, the Media Server is responsible for storing them in the vector database. The advantage of this approach is that the embeddings are deeply integrated into the solution, and one can assume that this is the faster approach in terms of performance because Video Endpoints are hardware-specific products that can implement the embedding operation through dedicated chips. The disadvantage is that it requires hardware-specific components; thus, the solution becomes more expensive than the other approaches.

Finally, the fourth approach, represented in Figure 3.4, assumes, as the first approach, an Encoding Server, but in this case, no streaming session is established with the Media Server; instead, the Encoding Server gets the video data from the video database and encodes the embeddings. The advantage of this approach is that the embedding generation can be performed *offline*. The disadvantage is that some video surveillance systems may require a real-time embedding generation, for example, a CCTV solution that creates an alarm whenever an embedding is classified as anomalous.

On this occasion, we followed the last stages in the last approach. We passed the entire video dataset's frames through an encoding script that computed the corresponding embeddings. These were passed to a vector database. Despite the limitations described for this approach and as we already mentioned, in this Master Thesis, we did not seek to evaluate the ingestion process but the subsequent retrieval.

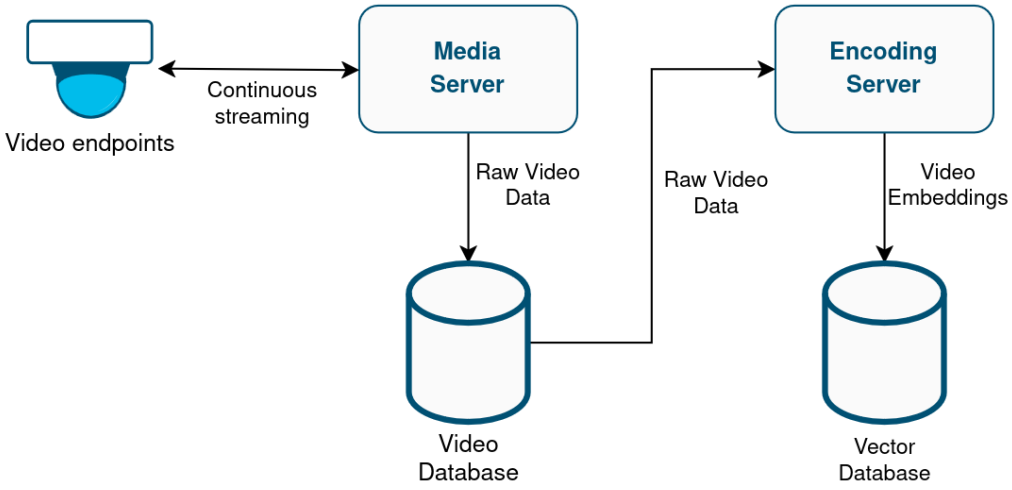

Figure 3.4: An external embedding server is responsible for *offline* encoding videos into embeddings.

3.1.2 Data storage

The data retrieval pipeline needs two databases to work properly: the video database and the vector database. Both of these databases are populated through the data ingestion pipeline, independently of the approach selected from the ones presented in Figures 3.1 to 3.4. All the details of these two databases are presented in Section 3.3.

The video database is deployed in a FTP which can be mounted in the file system of any machine. The videos are located in a specific folder and are identified just by its file name.

The vector database stores embedding representations of the frames stored in the video database generated from an encoding model. For each of the stored embedding, several metadata variables are defined: the video file name of the frame used to generate the embedding and the frame number, so each vector can be associated with its corresponding frame.

Another dataset of vectors stored in the vector database is the one generated from UCA. In this case, one embedding was generated for each of the UCA annotations by taking the centroid of all its frames' embeddings. The metadata stores the video file name, the start frame of the annotation and the end frame. However, for simplicity, in this introduction we will assume the vector database stores the data presented in the previous paragraph.

3.1.3 Data retrieval

Data retrieval encompass the process of accessing to data generated during the ingestion pipeline to perform some kind of analysis. This pipeline involves several factor that must be taken into account:

- **Accuracy**: The retrieved data contains the information that the user expected to receive. This involves filtering the irrelevant data and returning only the appropriate ones. As a metric, it focuses on the final result of the retrieval, regardless of the process involved.
- **Availability**: The data can be retrieved as it was stored within a bounded period. The goal is to quantify the quality of the experience in terms of access to the information.
- **Latency**: The data retrieval operation is performed fast enough to make the system usable. This figure of merit expands on the quality of the experience, addressing the time to respond.

As discussed in Section 2.2, one of the main flaws in traditional video surveillance is the need to review all video data while searching for relevant information manually. Video summarization is a great way to ease this task by automatically detecting relevant events along the video footage during the data retrieval pipeline. This project takes a step further and implements cross-modal embedding models to generate specific summaries from text prompts written by the user.

Our approach for the data retrieval application is illustrated in Figure 3.5. There we include the components that form the application:

- **Client**: it is the user responsible for reviewing the surveillance videos. In classic CCTV, this is done manually. With the approach described in this Thesis, the reviewing process is substantially automated.
- **Video Summarizer**: is the main component of our application. It takes the prompt from the client and orchestrates the communication with other components to create the summary to be returned to the client.
- **Encoding Server**: it is a minimal server that takes a piece of data (i.e.: text) and returns its vector representation or embedding. As we previously discussed, the application does not include the ingestion pipeline and videos had already been encoded and stored.
- **Vector Database**: stores the vector representations or embeddings of the video data obtained during the ingestion. These vectors retain the semantic representations of the original video data, so the database can be consulted with a query vector to perform similarity searches.
- **Video Database**: it is the storage area of the video footage from the cameras. The videos can be accessed using some metadata, for example, the ID of the camera or the video date. In the case of this project, the video database is just a FTP server mounted in the Video Summarizer file system.

The logical view of the procedure conducted to respond to a retrieval query is represented in Figure 3.5 through the enumerated arrows:

(1) The client sends a text query to the video summarizer.

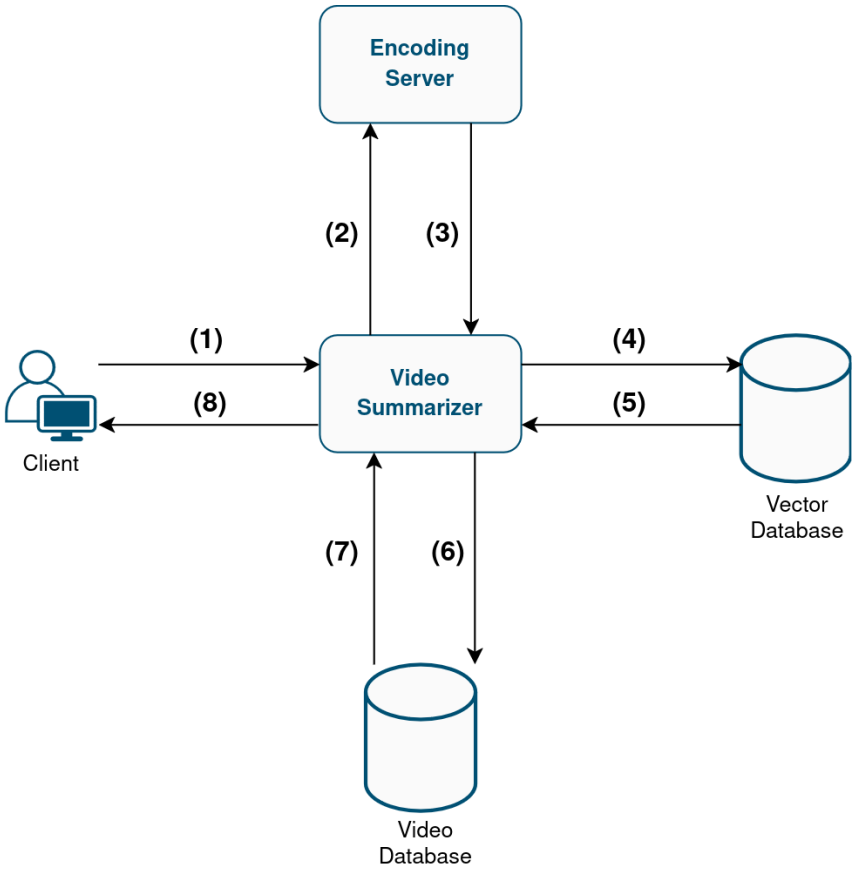

Figure 3.5: Components of the data retrieval and summarization pipeline and their communications. Numbers describe the communications carried to process a query since (1) it is communicated down to the point where (8) the result is provided.

(2) The video summarizer sends the text to the encoding server.

(3) The encoding server uses a cross-modal transformer for text (in this case, CLIP) and responds with the vector representation of that text.

(4) The video summarizer then uses that vector representation to query the vector database.

(5) The vector database responds with the metadata associated with the top-K most similar vectors. These vectors correspond to frames similar to the text query sent by the client, and the metadata consists of the necessary information to locate those in the video database. These frames are considered key-frames of events happening in the video.

(6) The proposed video summarizer uses the frames' metadata to locate the relevant videos and retrieve them.

(7) The video database returns the requested videos. The video summarizer creates the summaries by taking the key-frame and its surrounding frames.

(8) These summaries are returned to the client as a response to the query.

As mentioned, this description of the data retrieval pipeline is just an overview, introducing its components and procedures. From Section 3.3 onward a more detailed explanation of all of the elements listed here will be given.

3.1.4 Presentation

The application was implemented using web technology. The client interacts with a website through a standard web browser like Chromium or Firefox, which sends HTTP requests to a Web server deployed with NodeJS technology acting as the Video Summarizer, performing all the operations specified above. Details of the application will be presented in this section hereafter. Validation tests were covered in the browsers previously listed.

Figure 3.6 shows the presentations of the results of the retrieval application for the prompt: «a man lying on the ground». The steps taken to obtain those results were: (1) introducing the prompt in the search bar. (2) Selecting the options below the search bar (encoder, dataset and database). (3) Pressing the «Search» button or pressing the Enter key.

Below the options, the website presents a list of summaries. In Figure 3.6, only two can be seen, but the application outputs 10 summaries. Next to the video container, the name of the original video from UCF dataset is shown. For example, the summary shown in the Figure comes from the video `Assault018_x264.mp4`. The user can play all the resulting summary videos, which will stream from the Video Summarizer Server.

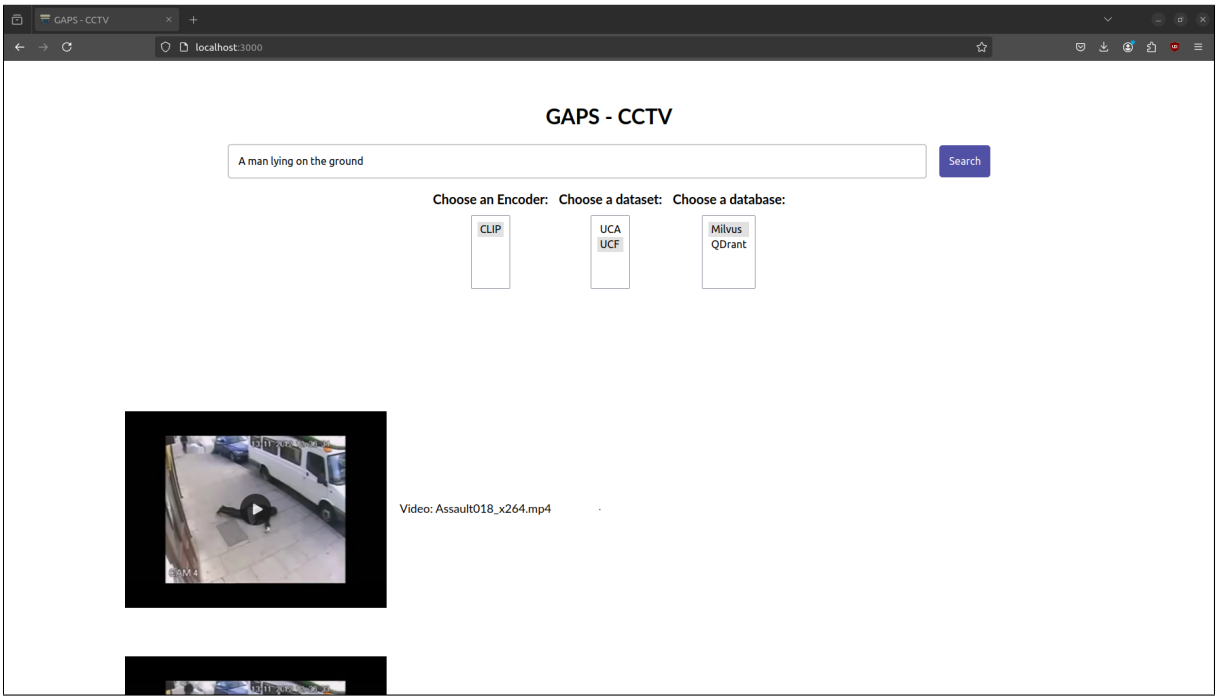

Figure 3.6: Screenshot of the application showing video summaries for the text «a man lying on the ground».

3.2 Architectural overview

As indicated at the beginning of the chapter, the general approach is to develop different components that are common in real-world solutions but with the ability to manage multimedia data through embeddings.

Following this idea, we identified several key requirements for the architecture selection:

- The developed components follow an application-independent capability, so the issues arising from these implementations can be tackled in a generalized way.
- Individually developed components must be independent of all others. Consequently, a thorough study can be performed without the noise coming from the undesired coupling of components.
- The system must follow a realistic and market-acquainted architecture, so the danger of falling into an *ad-hoc* solution with no realistic deployment possibilities in the real market is avoided.

From the study of these requirements, the multilayer architecture style was chosen. This style is widely used in the industry as it allows functional isolation of the different components. This style helps when a variety of different technologies are used at once.

In this architecture, the components are arranged in vertically stacked layers. Each layer presents an API to its immediate upper layer. These APIs are a set of methods that provide different functionalities that can be queried. It is up to the upper layer to use these methods in the most suitable way to achieve functionality. The most common implementation of this architecture is the 3-tier layered architecture. It presents, from top to bottom, an *application* layer, a *business* layer (with a sub-layer responsible for embedding operations), and a *data* layer.

Figure 3.7 illustrates the layers in our system. As can be seen, the typical 3-tier architecture has been used. An interaction from the client will trigger the application layer to start calling the methods provided by the business layer. Each method belonging to the business layer will (or may) call the methods provided by the data layer. During this process, the upper layer has no idea how the lower layer's methods are implemented; it is just interested in getting the requested response, thus achieving a profound isolation between layers.

- *Data* layer: this layer works with two types of data: the surveillance video data and the cross-modal representations of those videos. Its job is to store these components so an effective retrieval is achieved, alongside ensuring their integrity. The vector database and video database belong to this layer.
- *Embedding* sublayer: this is a special kind of layer. As shown in Figure 3.7, its methods can only be accessed by the business layer. The main purpose of this layer is to compute embeddings of multimedia data and return them to the caller. The embedding sublayer was defined as a different layer rather than just another functionality of the business layer because isolation and independence of this component were also desired. The embedding server belongs to this layer.

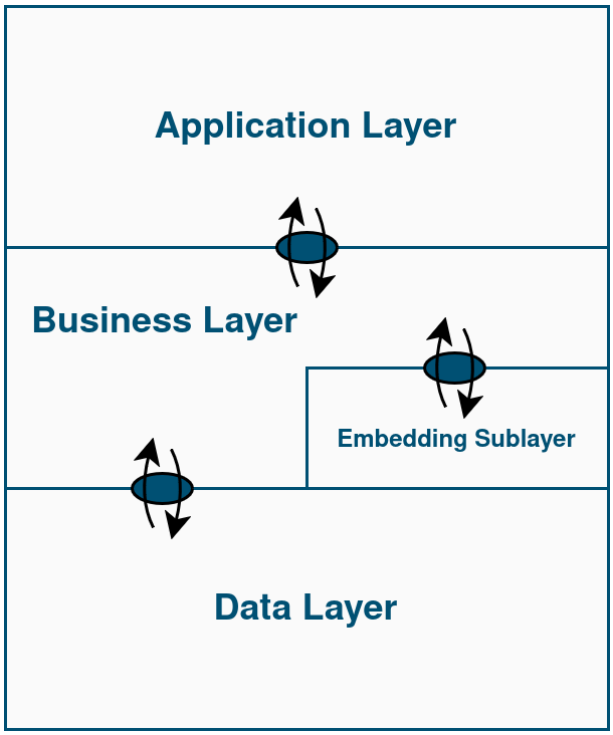

Figure 3.7: Different layers of the video retrieval application.

- *Business* layer: as for this implementation, this layer performs two main tasks:
  - It receives text prompts from the client, requests their vector representation to the embedding layer, and queries the result to the vector database, obtaining a list of references to the most similar video clips. This is to be served to the client.
  - It receives the requests from the client to form the video clips, specifying the video name and the start and end frame. This layer crops the video to a clip and sends it as a response.

  The video summarizer belongs to this layer.
- *Application* layer: this layer calls the methods the business layer provides according to the user's interaction with the application. It is also responsible for displaying the results of the methods to the user through a web browser. This layer is implemented by the web browser of the client.

The remainder of this chapter studies these layers and discusses our implementations.

3.3 Data layer

3.3.1 Video data

The data retrieval application works with the video data already ingested and stored in a video database whose elements can be obtained through an identification information

(like the camera identifier, the footage date, etc.). As mentioned earlier, due to the lack of real cameras, a video surveillance dataset called UCF-Crime [37] was used to simulate the video data.

In this Master Thesis, no specific management system was deployed for video data. Instead, a FTP server was used, with a directory storing each of the 1900 videos. Each video file is named with the category it belongs to, followed with a numeric identifier. Then, «`x264`» code is appended, which corresponds to the video codec H.264. An example of the name of one UCF video is «`Assault012_x264.mp4`». Figure 3.8 shows some of the UCF videos inside the FTP server.

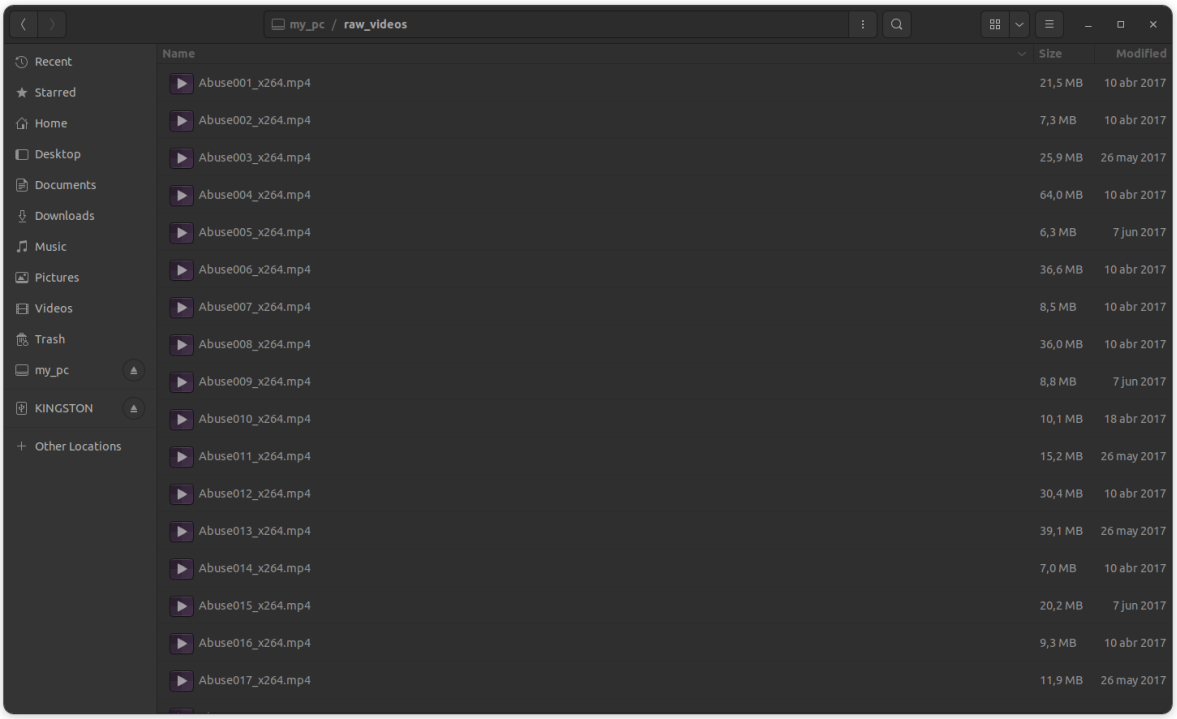

Figure 3.8: Screen capture on the folder of the FTP server containing all the UCF videos.

In the next subsection, we will explain how this video data was transformed into vector data by passing each frame through CLIP Visual Transformer and stored in a vector database, so semantic queries can be used to retrieve portions of this video footage and to generate appropriate summaries.

In the vector generation stage, we also used the UCA annotations over UCF dataset. UCA is interesting in this study because to generate event-level vectors, instead of at frame-level, by computing the embedding of each frame inside the event and then taking the centroid. This last centroid should aggregate all the information present in each frame. This approach reduces the size of the resulting vector dataset. If the accuracy of the data retrieval system is not affected, this presents a beneficial approach in terms of storage. The disadvantage is that a previous event-based segmentation of the video footage has to be covered, which is not tackled in this project.

3.3.2 Vector data

The retrieval application uses vector data to build a semantic-aware retrieval and summarization system. A robust and efficient vector management system is vital. In this Thesis, two vector databases were deployed: Milvus [38] and Qdrant [39]. Additionally, we performed a study on how a traditional relational database like PostgreSQL could be extended so it supports storage for cross-modal vector data.

3.3.2.1 Vector dataset generation

For the vector data generation, we used the two datasets: UCF and UCA.

The vector data generated from UCF videos took the following procedure:

1. Open each video file from UCF dataset using OpenCV library [40].
2. Read the next frame from the video.
3. Pass the frame through CLIP Vision Encoder[1].
4. Save the resulting embedding in disk as a `.npy` (NumPy [41]) file whose name indicates the original video name and the frame number.
5. Go back to step 2 until video is done.
6. Go back to step 1 until all videos are done.

In the vector database, each of these resulting embeddings were stored alongside the video file name and the frame number as metadata, so each embedding could be tracked to the original frame from which it was generated. Figure 3.9 illustrates how UCF videos were encoded and what was finally stored in the vector database.

This process resulted in a dataset of 12,247,318 embeddings[2], totalling a size of 13.14 GB. To save time during the ingestion process of this dataset, a NumPy array was built from the individual files with each row containing an embedding and some metadata, specifically, the source video file name, the frame number and the video path. Then, the array was partitioned in half and stored separately, creating two files: «`p0-full-ucfclipframes.npy`» and «`p1-full-ucfclipframes.npy`». Then, these files were uploaded to both vector databases inside a collection called «`ucfclipframes`».

Another vector collection was built from UCA dataset. The procedure to build this collection was the following:

1. Open a video file from UCA dataset using the OpenCV library.
2. Read the frames belonging to the next annotated event from the video.

[1]Actually, the script waited until 64 frames were read. Then, it passed this batch through CLIP encoder. This was only done to reduce the encoding time, and it is not mentioned for simplicity.

[2]There are less frames than the number shown in Table 2.2 because some videos had to be discarded due to bad formatting.

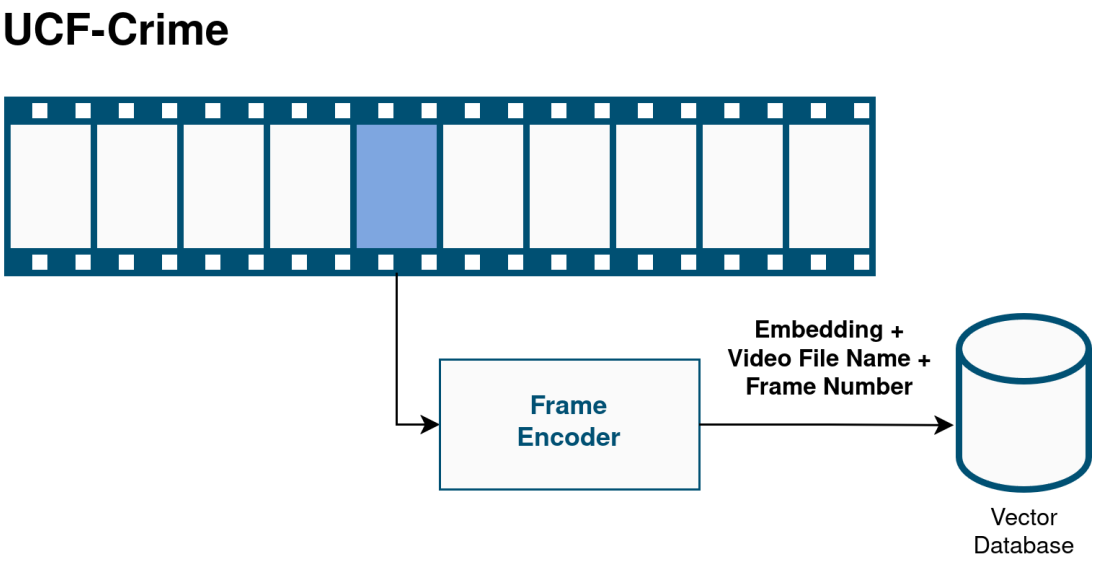

Figure 3.9: Encoding of UCF videos and storage of vector dataset.

3. Pass each of the frames through CLIP Vision Encoder.
4. Calculate the centroid embedding of the resulting set of embeddings.
5. Save the resulting centroid embedding in disk as a `.npy` file whose name indicates the original video and the event's start and end frame number.
6. Go back to step 2 until all events from the video are done.
7. Go back to step 1 until all the videos are done.

In the vector database, each of these resulting embeddings were stored as in the case of UCF, but this time the metadata contained the video file name, the start frame of the event and the end frame of the event, so each embedding could be tracked to the original UCA annotation from which it was generated. Figure 3.10 illustrates how UCA videos were encoded and what was finally stored in the vector database.

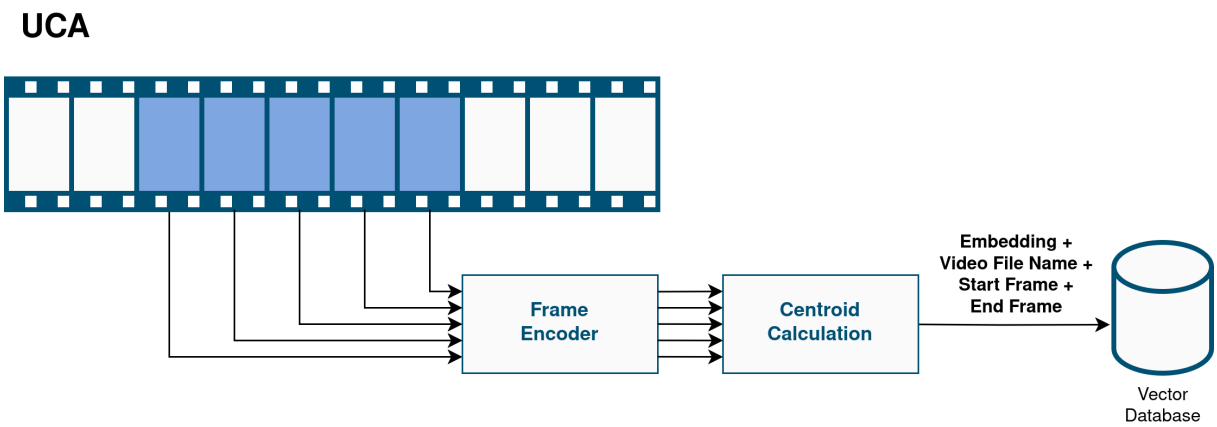

Figure 3.10: Encoding of UCA videos and storage of vector dataset.

During the encoding of this dataset, we identified a few clips of the UCA dataset with more than one annotation. For example, for the video `Abuse001_x264.mp4` the UCA

dataset provides two annotations starting in second `8.9` and ending in second `11.2`. The coding of these two annotations generates the same embedding, and there is no reason in this context to store duplicate embeddings, so duplicate annotations only have one corresponding embedding. Thus, after this encoding operation was finished, a collection of 22,356 vectors was obtained.

Similarly, as in the case of UCF dataset, UCA NumPy files were then uploaded to the vector databases inside a collection called «`ucaclipcentroid`».

The rationale behind this second encoding procedure is to test the semantic capabilities of CLIP model if it is only used for retrieval, being the summarization process already performed by UCA segmentation.

Now, we will presents the two vector database management systems deployed in this work.

3.3.2.2 Milvus

Milvus is a vector database developed at Zilliz. Milvus was developed to store high-dimensional vector representations of unstructured data along with some metadata attributes in large-scale applications. It is built on top of Meta's Faiss library for vector data management.

Milvus presents a set of interesting features: it allows a hardware specific computing acceleration and smart combination of CPU and GPU computing. It supports a wide range of indexes: IVF, HNSW, ANNOY, etc. Among its searching features, we can find: vector search (ANN search and radius search, which returns the vectors inside a specific area around the query vector), attribute filtering, multi-vector search and direct fetching using keywords.

Figure 3.11 summarizes the Milvus component architecture. As can be seen, Milvus implements a highly decoupled architecture, isolating the elements. An interesting feature is that the messages and inputs from the client go through an access proxy, which decides which component to call to obtain the desired results. Another interesting feature is the use of a server known as etcd that stores Milvus' internal metadata and internal services endpoints (in the form of key-value pairs) and an external tool for object storage (in the case of this project, Minio will be used). Both of these tools work transparently for the end user, so no configuration or operation considerations are needed.

Milvus offers three deployment modes: Lite, Standalone and Cluster. Milvus Lite is just a Python library with reduced capabilities used for rapid prototyping. Milvus Cluster is a cloud-native implementation designed for large-scale applications and deployed through Kubernetes. Finally, Milvus Standalone is a single-server deployment inside a Docker container. The last implementation is the most suitable for the application developed in this Master Thesis. Thus, we used it in our tests.

Once Milvus was up and running inside a Docker container, a Python script was used to feed the database with UCF embeddings. Listing 3.1 shows a simplified version of the script used. A for loop goes through each of the rows of the NumPy array containing all the embeddings and metadata and uploads the information to the database. All the embeddings were stored in a collection called «`ucfclipframes`». The same thing was done

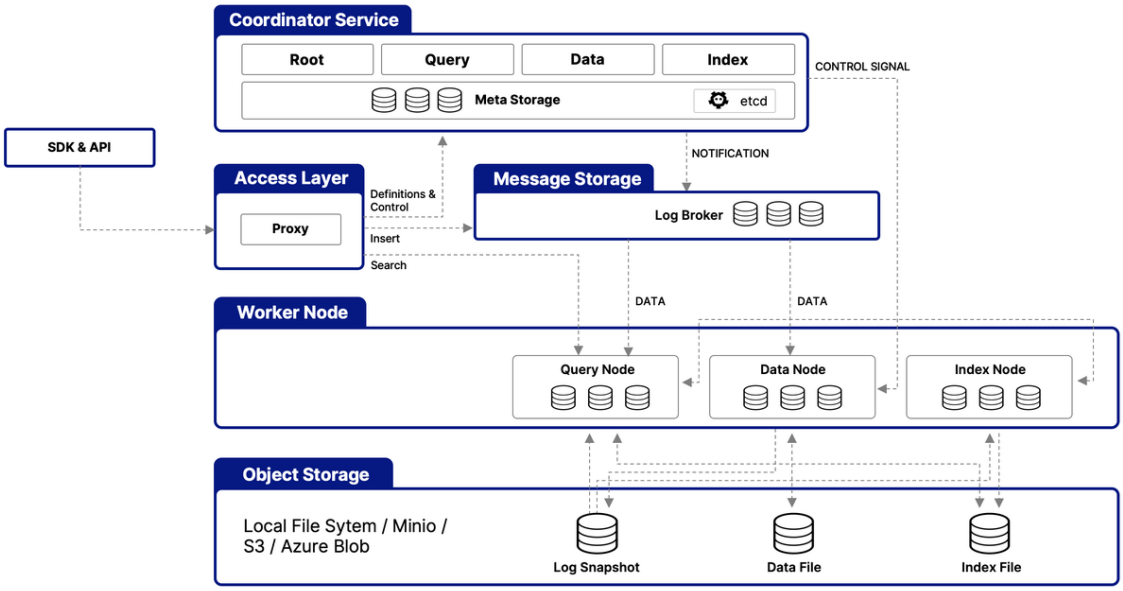

Figure 3.11: Milvus architecture. Extracted from [42].

with UCA datasets, storing them in a collection called «`ucaclipcentroid`».

```
database = DatabaseBuilder.build(database_name='milvus', encoder_params=
    encoder_params)

basename = 'full-ucfclipframes.npy'

for partition_id in ['p0-', 'p1-']:

    # Load partition array
    partition_file = os.path.join(DISK_PATH, partition_id+basename)
    df = np.load(partition_file, allow_pickle=True)

    # For each embedding
    for row in tqdm(df):

        # Upload to database along with metadata
        database.upload_embedding(
            id=row[ID],
            emb=row[EMB],
            metadata={
                'video': row[VIDEO],
                'frame_n': row[FRAME_NO]
            }
        )
```

Listing 3.1: Python script to insert the embeddings into Milvus database.

After the data ingestion, an index had to be built in order for the database to be queried. This can be done with the Python script like the one shown in Listing 3.2, where a IVF-PQ index is built for collection `ucfclipframes`.

```
from pymilvus import MilvusClient

# Connect
client = MilvusClient(uri=f'http://172.18.0.6:19530', token='root:Milvus
    ')
```

```
5
6  # Prepare params
7  index_params = MilvusClient.prepare_index_params()
8  index_params.add_index(
9      field_name='vector',
10     index_type='IVF_PQ',
11     metric_type='COSINE',
12     index_name='ucfclipframesivfpq',
13     params={
14         'nlist': 4096,
15         'm': 32
16     }
17 )
18
19 start = time.perf_counter()
20 # Create index
21 client.create_index(
22     collection_name='ucfclipframes',
23     index_params=index_params,
24     sync=True
25 )
```

Listing 3.2: Index building in Milvus

The reason behind the selection of IVF to index the UCF dataset was because we followed the index selection criteria proposed by Faiss [20] and described in Section 2.4.1.4. The parameter `nlist` defines the number of clusters to divide the vector dataset in, and `m` defines the number of sub-vectors used. These two parameters were chosen following the guidelines of [43], and the selection was `4096` for `nlist` and `32` for `m`.

We followed the same process to build an index for UCA dataset. According to the Faiss criteria, the most suitable index was a HNSW index. HNSW requires two parameters to be built: `M`, which defines the maximum number of outgoing connections in the graph per each node and `efConstruction`, which controls the speed of the index construction. For these parameters' selection the same guidelines as in the UCF case were followed, and the values were `32` for `M` and `256` for `efConstruction`.

Once this process was completed, the Milvus database was ready to be tested and to experiment with the querying system.

3.3.2.3 Qdrant

Qdrant is another Vector Database Management System that has been in the market for some time. Qdrant is designed to support the storage of large datasets of high-dimensional vectors along their metadata and to be queried employing similarity searches. Its collections can be appropriately indexed to provide a fast and accurate search.

Qdrant presents a set of relevant features: allows hardware acceleration using SIMD parallel computing, presents a custom HNSW implementation, supports distributed and standalone deployment, allows vector compression and on-disk storage and supports hybrid searches. One thing to note is that it provides a great documentation, allowing fast up-and-running.

Figure 3.12 provides an application-level overview of the Qdrant database. Qdrant

implements the usual vector database pipeline. Qdrant gets its data from Machine Learning engineers during the ingestion pipelines, while the database is queried through its APIs during the retrieval pipeline.

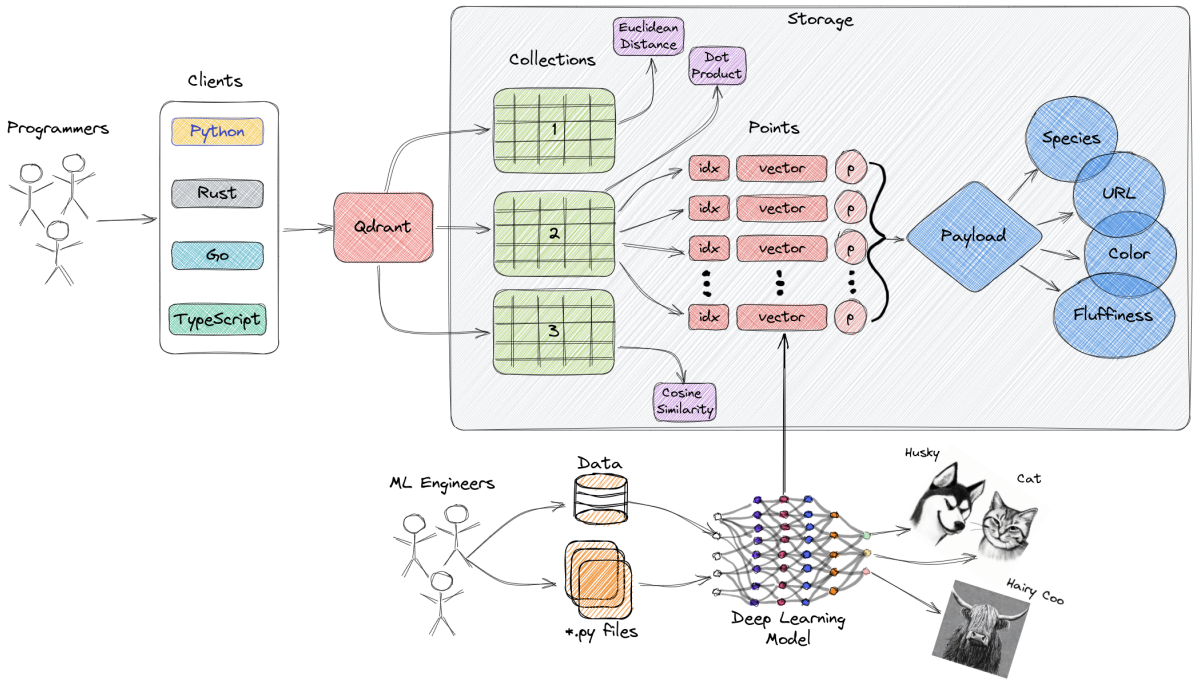

Figure 3.12: High-level overview of Qdrant architecture. Extracted from [39].

A unique feature Qdrant offers is a web interface where users can explore its vector collections. The web interface offers a scripting engine that can perform quick operations in the database and several graphic tools to visualize the vector space of the different collections.

Once Qdrant was up and running inside a Docker container, a Python script was used to upload the UCF embeddings to a collection called «`ucfclipframes`». The code used for Qdrant ingestion is similar to the one showed in Listing 3.1. Just changing the parameter `database_name` of the function `DatabaseBuilder.build` from `'milvus'` to `'qdrant'` is enough, thanks to the database handler class implemented in this work. UCA embeddings were also uploaded to a collection called «`ucaclipcentroid`».

In terms of indexes, Qdrant only offers one index for its collections: a custom implementation of HNSW. This index is managed entirely by the VDBMS and no operation is required by the client. The index update is also managed internally by Qdrant.

3.3.2.4 PostgreSQL extension

Before finishing the section related to the data layer, a brief overview of PostgreSQL extensions for vector data will be given. Section 2.4 mentioned that providing vector database features to widely spread relational databases such as PostgreSQL without losing much performance would significantly impact the market. Many companies could rely on well-known storage technologies to migrate their apps to an embedding-oriented design.

PostgreSQL [44] is a well-established relational database that has been in development

for 35 years and has demonstrated its outstanding performance in terms of reliability, data integrity, robustness and ACID-compliance. Another powerful tool of special interest for this work is PostgreSQL's great extendability, which allows programmers to fine-tune PostgreSQL to their specific requirements. As for this Thesis, a PostgreSQL extension to make the system work like Milvus or Qdrant may be of interest.

This project, however, has not developed a full PostgreSQL extension to manage vector data due to the complexity of the task and the fact that other vector extensions already exist. Only a brief proposal on how vector data could be modelled is given, along with the corresponding code, but no search index has been manually developed.

There exist two well-known PostgreSQL extensions that allow the database to store and query high-dimensional vector data: pgvector [31] and PASE [32]. pgvector is an extension developed mainly by a programmer called Andrew Kane and still recieves updates in its Github repository. PASE is an extension developed by Alipay, a Chinese tech company. It is older than pgvector and has not received updates in its GitHub repository for 3 years.

Both extensions define vectors as a custom data type: a frame that stores a variable-length float array along the dimension number as header data. In conjunction with this type definition, the extensions define the functions needed by PostgreSQL to ingest the vector data from a SQL query string and output the data to a response string. Vector comparison operators such as L^2 norm or cosine distance are also defined. Finally, both solutions implement IVF and HNSW indexes following the PostgreSQL index interface.

The extension developed in this Master Thesis follows this approach, defining a custom data type, adding extra parameters, all of fixed-length. Figure 3.13 represents the vector data type defined in our extension, and Listing 3.3 shows the implementation in C code.

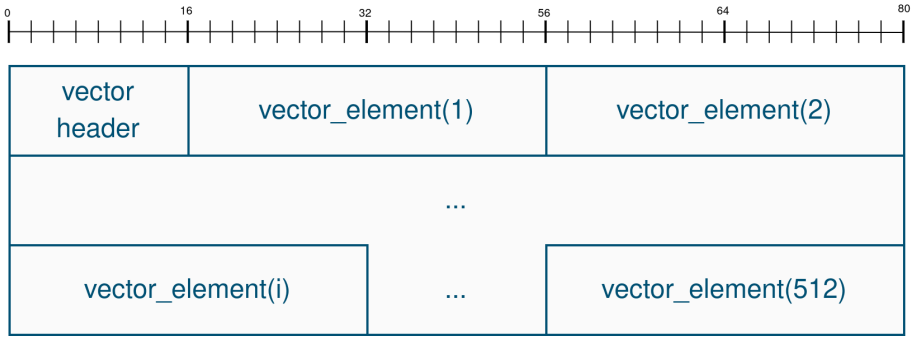

Figure 3.13: Frame structure of the vector data type.

```c
#define VECTOR_DIM_512 512

typedef struct VectorHeader
{
    uint8 metric;
    uint8 model_id;
} VectorHeader;

typedef struct Vector
{
    VectorHeader header;
    float4 x[VECTOR_DIM_512];
} Vector;
```

Listing 3.3: Definition of vector data type.

The vector data starts with two bytes as header information. The first is an ID for the metric that must be used to compute similarities with the vector. This data has been "hard-coded" inside the data type because it would make no sense to compare two CLIP-generated vectors with metrics other than cosine distance. The second is also an ID that indicates which model generated the vector. Again, comparing two vectors encoded from a different model would make no sense. The mapping between the IDs and the different metrics and models is left to the users. After the header, the vector data begins as 4-byte float point numbers.

The selection of 512 as the dimension of the vector is chosen because in this Master Thesis, only CLIP Base-32 is used, which produces a vector of 512 elements. However, it is possible to define a variable length vector (it is the approach taken by pgvector and PASE). However, for this case, the fixed-length vector was preferred, as a variable vector means more complicated and less readable code, and, since the intention is not to create a full vector extension, it was decided to preserve code simplicity and readability.

Aside from defining custom data types, the PostgreSQL extension framework allows the developer to create custom operators to be used over the custom data. In the case of this project, the cosine distance operator was developed, taking as a reference the implementation done in pgvector. The code of this operator can be seen in Listing 3.4. The function `cosine_distance` is taken literally from pgvector code and computes the metric between two generic float vectors. The second function is a wrapper for the 512-vector defined. First, it calls the function `CheckModel` to check if the two vectors have the same header (otherwise, the comparison would have no meaning), and if they do not, an error is sent. Otherwise, the cosine similarity is computed and returned.

```
1  // This is exactly from pgvector
2  static float8
3  cosine_distance(float4 * a, float4 * b, uint32 dim)
4  {
5      float similarity = 0.0;
6      float norma = 0.0;
7      float normb = 0.0;
8
9      /* Auto-vectorized */
10     for (int i = 0; i < dim; i++)
11     {
12         similarity += a[i] * b[i];
13         norma += a[i] * a[i];
14         normb += b[i] * b[i];
15     }
16
17     /* Use sqrt(a * b) over sqrt(a) * sqrt(b) */
18     return (float8) 1 - (similarity / sqrt((double) norma * (double)
       normb));
19 }
20
21 FUNCTION_PREFIX PG_FUNCTION_INFO_V1(cosine_distance512);
22 Datum
23 cosine_distance512(PG_FUNCTION_ARGS)
24 {
25
26     Vector * v1 = (Vector *) PG_GETARG_POINTER(0);
27     Vector * v2 = (Vector *) PG_GETARG_POINTER(1);
28     float8 sim;
```

```
29
30      CheckModel(v1->header, v2->header);
31      sim = cosine_distance(v1->x, v2->x, VECTOR_DIM_512);
32
33      PG_RETURN_FLOAT8(sim);
34  }
```

Listing 3.4: Cosine distance operator implementation in PostgreSQL extension framework

PostgreSQL extensions require the use of many macros (`PG_FUNCTION_INFO_V1()`, `PG_GETARG_POINTER()`, `PG_RETURN_FLOAT8()`, etc.). This is required so the PostgreSQL core engine can manage the data and code of these functions coherently.

Listing 3.5 shows the SQL code needed to link all the C code to the PostgreSQL core engine so it can start using the data and functions defined. Lines 5 to 37 define the vector data type and link all the functions related to the data type input-output procedures. Lines 41 to 53 link the cosine distance function and assigns an operator, `<=>`, that can be used in SQL queries.

```
1  \echo Use "CREATE EXTENSION pgsurv" to load this file. \quit
2
3  --- Define temporal vector type
4
5  CREATE TYPE vector;
6
7  --- Link the I/O functions
8
9  CREATE FUNCTION vector_in(cstring)
10     RETURNS vector
11     AS 'MODULE_PATHNAME'
12     LANGUAGE C IMMUTABLE STRICT;
13
14 CREATE FUNCTION vector_out(vector)
15     RETURNS cstring
16     AS 'MODULE_PATHNAME'
17     LANGUAGE C IMMUTABLE STRICT;
18
19 CREATE FUNCTION vector_recv(internal)
20     RETURNS vector
21     AS 'MODULE_PATHNAME'
22     LANGUAGE C IMMUTABLE STRICT;
23
24 CREATE FUNCTION vector_send(vector)
25     RETURNS bytea
26     AS 'MODULE_PATHNAME'
27     LANGUAGE C IMMUTABLE STRICT;
28
29 --- Create the type and link the I/O functions
30
31 CREATE TYPE vector (
32     INTERNALLENGTH = 2052,
33   INPUT      = vector_in,
34   OUTPUT     = vector_out,
35   RECEIVE    = vector_recv,
36   SEND       = vector_send
37 );
38
39 --- Link cosine distance function
```

```
40
41 CREATE FUNCTION cosine_distance512(vector, vector)
42     RETURNS float8
43     AS 'MODULE_PATHNAME', 'cosine_distance512'
44     LANGUAGE C IMMUTABLE STRICT;
45
46 --- Create SQL operator
47
48 CREATE OPERATOR <=> (
49     LEFTARG = vector,
50     RIGHTARG = vector,
51     FUNCTION = cosine_distance512,
52     COMMUTATOR = '<=>'
53 );
```

Listing 3.5: SQL code used to link the extension to PostgreSQL core.

Once all the C code is developed, a proper Makefile has to be constructed. For this matter, PostgreSQL offers a building infrastructure, but some variables have to be defined before. For this project, the Makefile was built based on the PostgreSQL documentation examples and on pgvector Makefile.

To finish with PostgreSQL extension, Listing 3.6 shows the Dockerfile to build an image with this extension included in a PostgreSQL instance.

```
1  ARG PG_MAJOR=17
2  FROM postgres:$PG_MAJOR
3  ARG PG_MAJOR
4
5  RUN apt-get update && \
6      apt-mark hold locales && \
7      apt-get install -y --no-install-recommends build-essential
       postgresql-server-dev-$PG_MAJOR
8
9  COPY . /tmp/pgsurv
10
11 COPY init.sql /docker-entrypoint-initdb.d/init.sql
12
13 RUN cd /tmp/pgsurv && \
14     make clean && \
15     make OPTFLAGS="" && \
16     make install && \
17     rm -r /tmp/pgsurv && \
18     apt-get autoremove -y && \
19     rm -rf /var/lib/apt/lists/*
20
21 RUN mkdir /data && \
22     chown postgres:postgres /data
```

Listing 3.6: Dockerfile to build an image of PostgreSQL with vector extension

3.4 Business layer

The business layer receives the client's text prompt, processes it, and triggers the procedures. It takes data from the data layer and returns the trimmed videos. These

are the summaries of the CCTV footage attending to the input text prompt. In this application, this layer is implemented by the Video Summarizer and the Encoding Server, which implements the embedding sublayer.

The Video Summarizer is the main component of the layer, and it is divided into two pipelines: one only for retrieving the video summaries metadata (video file name, start and end frame) and other to send the actual summaries.

The first pipeline is described with the following steps:

1. The server receives the client's text prompts.
2. It uses the Encoding Server to transform text data to embeddings.
3. It queries the vector databases, returning key frames metadata if UCF dataset is used or the summaries metadata if UCA dataset is used.
4. It generates the response with all the resulting video summaries, including the video file name, the start frame and the end frame, but does not stream the videos, just sends the metadata.

The second pipeline covers the following steps:

1. The server receives from the client information of one summary (video file name, start frame and end frame).
2. It trims the video to generate the summary.
3. It streams the video summary.

Note that the second pipeline's input is the output of the first pipeline. There are two main reasons for dividing the pipeline in two: one is to provide more isolation to the different procedures, avoiding mixing database querying with video processing; the second is to allow the client to request each of the video summaries with a different HTTP request so, if one video trimming or download process fails, it does not affect the rest.

This division is not shown in Figure 3.5, where the response to the client first request is supposed to be the actual video summaries.

The existence of two pipelines with calls to other components creates a great heterogeneity within the Video Summarizer's functions, which requires the server to provide a high degree of abstraction with no coupling between procedures, so all the services can be provided reliably. A web architecture that fulfills all these matters is known as the Representational State Transfer (REST).

REST defines a set of principles relevant to our requirements. First, it is stateless, that is, each client-server interaction is independent and it is not affected by past interactions. Second, it is based on HTTP methods, exploiting the protocol technology to its full potential. Finally, it is resource-based, that is, procedures are accessed through predefined URIs (referred on this document as routes), and results are returned in a common format easily understood by clients (in this Thesis' implementation, JSON).

Among all the possible frameworks to implement REST APIs, we chose Node.js [45], a Javascript execution environment. The reasons behind using Node.js for this application are the following:

- Node.js is a widely spread solution for backend servers and REST services. At the beginning of this Chapter, we emphasized our aim to build on common technologies used in the real world.
- Node.js is a suitable tool for implementing a REST API due to the powerful library environment offered, highlighting Express.JS package, which is used to create web applications in Node.js.
- The client side of the application uses Javascript language to fetch the server, and Node.js is also programmed in Javascript. Having the same language on the client and the server side brings clarity to the project.
- The author of this Master Thesis has prior experience with Node.js language, so the development process was easier and faster.

Figure 3.14 provides an overall view of the general folder architecture of a Node.js application. In the root folder, the main Javascript file is located, which starts the application. When needed, a folder stores the static content to be rendered or executed in the client's machine. Another folder stores the Javascript code for the different routes of the API. This is the structure we followed for this project.

Listing 3.7 shows the Dockerfile of the Video Summarization Server. Listing 3.8 shows the deployment under Docker Compose of the Video Summarization server. Line 9 shows that the server must have access to a folder with UCF dataset videos. This is used for trimming the video.

```
1  FROM node:18-alpine
2
3  # Set the working directory
4  WORKDIR /usr/src/app
5
6  # Install the dependencies
7  RUN npm install express
8  RUN npm install fs path dotenv
9  RUN npm install fluent-ffmpeg ffmpeg-static
10 RUN npm install @ffprobe-installer/ffprobe
11 RUN npm install @zilliz/milvus2-sdk-node@2.4.10
12
13 # Copy the app
14 COPY . .
15
16 # Expose port
17 EXPOSE 3000
18
19 # Start the server
20 CMD [ "node", "app.js" ]
```

Listing 3.7: Video Summarization Dockerfile

```
1  services:
2      node-server:
```

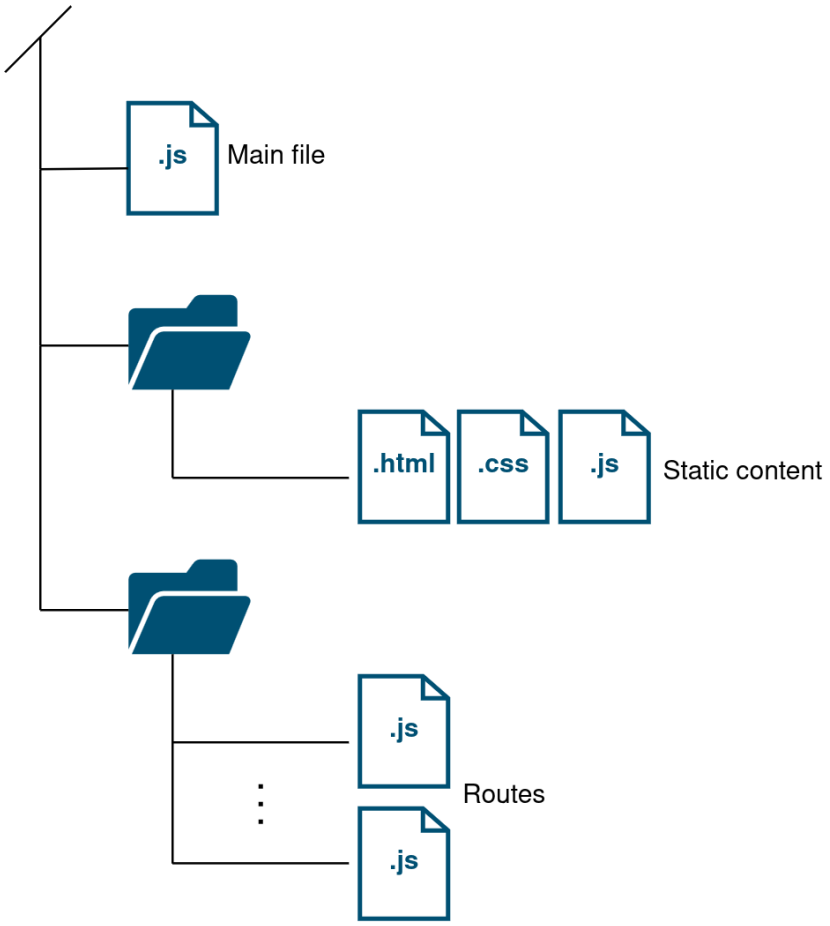

Figure 3.14: Folder structure of a typical Node.js web app project.

```
3         build:
4             dockerfile: Dockerfile
5             context: ./node-server
6         ports:
7             - ${NODEJS_PORT}:${NODEJS_PORT}
8         volumes:
9             - ${UCF_VIDEOS_PATH}:/videos
10        depends_on:
11            - embedding-server
```

Listing 3.8: Video Summarization Server deployment in docker-compose file.

3.4.1 Serving static content

Before any of the REST routes can be called, the Server sends the static web content. That is, the HTML, CSS and Javascript files to any client that access the Server URI. These files are managed by the client's web browser as we describe in Section 3.5.

For now, the only interesting fact is that this static web content allows the client to start making HTTP requests to the actual REST API.

3.4.2 Encoding text prompts, querying the database and returning video summaries metadata

This subsection describes the first pipeline from the ones specified at the beginning of the section.

The whole data retrieval procedure is triggered when the client introduces a text prompt in the application. This text prompt is sent to the Video Summarization Server, and this last sends it to the Encoding Engine, which transform the text into an embedding.

To provide this service, the REST API defines a route, `/query`, with two subroutes: `/query/milvus` and `/query/qdrant`, both accessed with the `GET` method of HTTP. The client uses either route, depending on the database it wants to use. Along the HTTP `GET` request, the client sends three parameters:

- `text`: it is the actual text prompt written by the client.
- `encoder`: specifies the text encoder the client wants to use to generate the text embedding.
- `dataset`: it specifies the dataset (database) to be searched UCF or UCA.

The whole process is illustrated in Figure 3.15. Details of the information sent to the Encoding Server will be presented in Section 3.4.4. We only need to acknowledge that it returns an array with the embedding of the text prompt generated with the encoder specified in the client request.

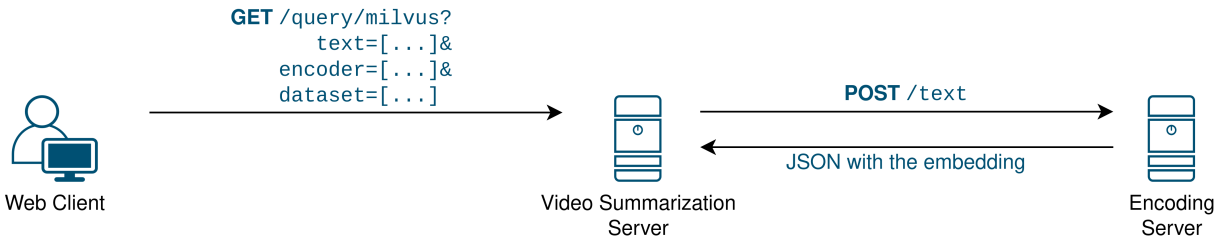

Figure 3.15: The client sends a request introducing the text prompt and other parameters. The Summarization Server uses the information introduced in the request to obtain the text embedding accordingly.

We use a text array to query the vector database specified in the route. This procedure depends on the API provided by the vector database: Milvus provides a specific API for Node.js, while Qdrant requires a HTTP `POST` request to be built.

Moreover, on the dataset parameter to be used in the retrieval process:

- If UCF: the collection `ucfclipframes` is queried with the embedding of the text prompt. We obtain a metadata response containing the video file name and frame number corresponding to the 10 keyframes with the embeddings that scored the highest similarity to the text prompt.
- If UCA: the collection `ucaclipcentroid` is queried with the embedding of the text prompt. We obtain a metadata output containing the 10 UCA events whose

embeddings scored the highest similarity to the text prompt. The results include the name of the video file, the starting frame number, and the end frame.

The video summary definition also depends on the selected dataset for the query:

- If UCF: the summaries are built by taking the surrounding frames to each returned keyframe. It was decided that a proper summary could be built from the start frame at 75 frames before the keyframe and the end frame at 75 frames after the keyframe.
- If UCA: the summary is the event defined in the UCA dataset.

The response sent to the client is a JSON array containing all the summaries information. That is, the video file name, the starting frame and the end frame.

3.4.3 Generating video summaries

This subsection describes the second pipeline from the ones specified at the beginning of the section.

The client obtains the actual video summaries by sending a HTTP request with the summary information. The Video Summarization Server then accesses the corresponding UCF video in the video database (the FTP server), trims the video accordingly, and streams the result to the client.

For this service, the route `/streaming` is specified, so the client can send a HTTP `GET` request along the following parameters:

- `name`: it is the UCF video file name.
- `startFrame`: specifies the start frame of the summary.
- `endFrame`: specifies the end frame of the summary.

To trim the videos, we used the FFMPEG library. The procedure is as follows:

1. *Validate start and end frames.* If start frame is less than 0, 0 is chosen as start frame. If end frame is greater than the video length, the video length is chosen as end frame. The video length is obtaied using FFProbe, a tool provided by FFMPEG library to gather information about video files.
2. *Obtain the video framer-rate* using FFProbe.
3. *Match start and end frames* into start and end seconds.
4. Use FFMPEG to *read the video and trim* it from the start second to the end second.
5. Use FFMPEG to *stream the video* to the client.

Listing 3.9 shows the actual code that was used to implement the previous procedure.

```
// Get video length
const videoLength = await getVideoLength(videoPath);

// Validate start and end frames
startFrameNumber = (startFrameNumber < 0) ? 0 : startFrameNumber;
endFrameNumber = (endFrameNumber > videoLength) ? videoLength :
    endFrameNumber;

// Get the video frame rate
const frameRate = await getVideoFrameRate(videoPath);

// Convert frame numbers to time (seconds)
const startSeconds = startFrameNumber / frameRate;
const endSeconds = endFrameNumber / frameRate;

// Use ffmpeg to trim and stream the video
ffmpeg(videoPath)
    .setStartTime(startSeconds)
    .setDuration(endSeconds - startSeconds)
    .outputOptions('-movflags', 'frag_keyframe+empty_moov')
    .toFormat('mp4')
    .on('error', (err) => {
        console.error('Error processing video:', err);
        res.status(500).send('Error processing video');
    })
    .pipe(res, { end: true }); // Stream
```

Listing 3.9: Video trimming inside the Video Summarization Server

3.4.4 Embedding layer

This layer is in charge of producing vector embeddings from multimedia data. The idea was to build a generic encoding server. This was not dependent on CLIP and could be extended in the future so that new models could be deployed, and different kinds of multimedia data could be encoded.

We developed a Flask server [46] for this layer. Flask is a minimalistic web app framework to build REST APIs in Python quickly and easily. The main reason to use Flask instead of any other solutions lies in Flask's integration with the Python language, which is the primary tool used to implement Deep Learning models.

As any other REST server, Flask can implement a series of HTTP routes to trigger the execution of a piece of code and return the results of that execution to the client. In the case of this application, only one route was implemented, `/text`, that takes a specific encoder model name and a text prompt and returns the embedding representation of the text obtained through the specified encoder. It was decided that, for privacy matters, this procedure will only respond to `POST` requests, so, the input information must travel in the request body in the form of JSON data. An example of the request payload can be seen in Listing 3.10. This request would cause the CLIP language model to encode «`example of a text prompt`» and the server to return the embedding as a response.

```
{
    "encoder": "clip",
    "data": "example of a text prompt"
```

```
}
```

Listing 3.10: Example of a payload expected in the POST request to encode a text.

Listing 3.11 shows the Python code used to implement the `/text` route of the API. First, the server checks whether the request payload includes all the data shown in Listing 3.10. Then, in line 17, the model specified by the user is built and loaded into memory using an encoder builder which follows the factory pattern design; that is, it hides the actual implementation of the model and provides a common interface to encode text. In line 20, the text is transformed into an embedding, and, finally, in line 25, if everything is executed correctly, the embedding is returned to the client.

```
# This route takes a text and responds with the embedding
@app.route('/text', methods=['POST'])
def get_text():

    # Check if request includes all the necessary data
    if (not check_request(request.json)):
        return 'Bad request', 400

    # Logging info
    print(f'Text encoding with: {request.json["encoder"]}')

    # Take the requested text
    text = request.json['data']

    try:
        # Build the selected model
        model = EncoderBuilder().build(request.json['encoder'])

        # Encode text
        emb = model.encode_text(text)
    except MemoryError:
        return 'ERROR: CUDA out of memory', 500

    # Return succesful response
    return emb.tolist(), 200
```

Listing 3.11: Route of Flask Server used to encode a text prompt.

The Dockerfile to run this server is shown in Listing 3.12.

```
FROM pytorch/pytorch:2.4.1-cuda12.4-cudnn9-runtime
WORKDIR /app

COPY requirements.txt .
RUN python -m pip install -r requirements.txt

COPY . /app

ENV FLASK_APP=app.py
ENV FLASK_ENV=development
ENV FLASK_RUN_PORT=1809
ENV FLASK_RUN_HOST=0.0.0.0

EXPOSE 1809

```

```
CMD ["flask", "run"]
```

Listing 3.12: Dockerfile to run the encoding server.

The lines to deploy the encoding server through Docker compose can be seen in Listing 3.13.

```
services:
    embedding_server:
        build:
            dockerfile: Dockerfile
            context: ./embedding-server
        ports:
            - ${EMB_ENGINE_PORT}:${EMB_ENGINE_PORT}
        volumes:
            - ${VCLIP_WEIGHTS_PATH}:/weights
        deploy:
            resources:
                reservations:
                    devices:
                        - driver: nvidia
                          count: all
                          capabilities: ["gpu"]
```

Listing 3.13: Encoding Server deployment in docker-compose file.

3.5 Application layer

The application layer has two responsibilities: (1) presenting the application's graphic interface to the user through a web browser and (2) calling the methods provided by the business layers (the routes defined in the Node.js Video Summarization Server) to satisfy user requests.

The general idea behind this layer is to provide a user-friendly interface. Then the user can interact with the application easily without worrying about the complexity behind the requests. The best way to achieve this goal is to develop a browser application that can be executed in a wide range of web browsers (Chrome, Firefox, Opera, etc.) through the combination of HTML, CSS and Javascript technologies.

HTML code is responsible for structuring the page and defining the elements that will be shown. CSS apply styles to the HTML elements (colours, sizes, margins, positions, etc.). Finally, Javascript code is responsible for the asynchronous procedures of calling the business layer methods whenever the client interacts with the page.

Figure 3.16 shows the developed webpage. It presents a set of video summaries obtained from the text prompt introduced in the search bar. Before the search, the user had to introduce, in addition to the text prompt, a series of options:

- Encoder: refers to the model that computes the text embeddings. The only option here is CLIP, but other options can be included.
- Dataset: refers to the video dataset used, UCF or UCA. This option determines the collection queried by the Video Summarization Server.

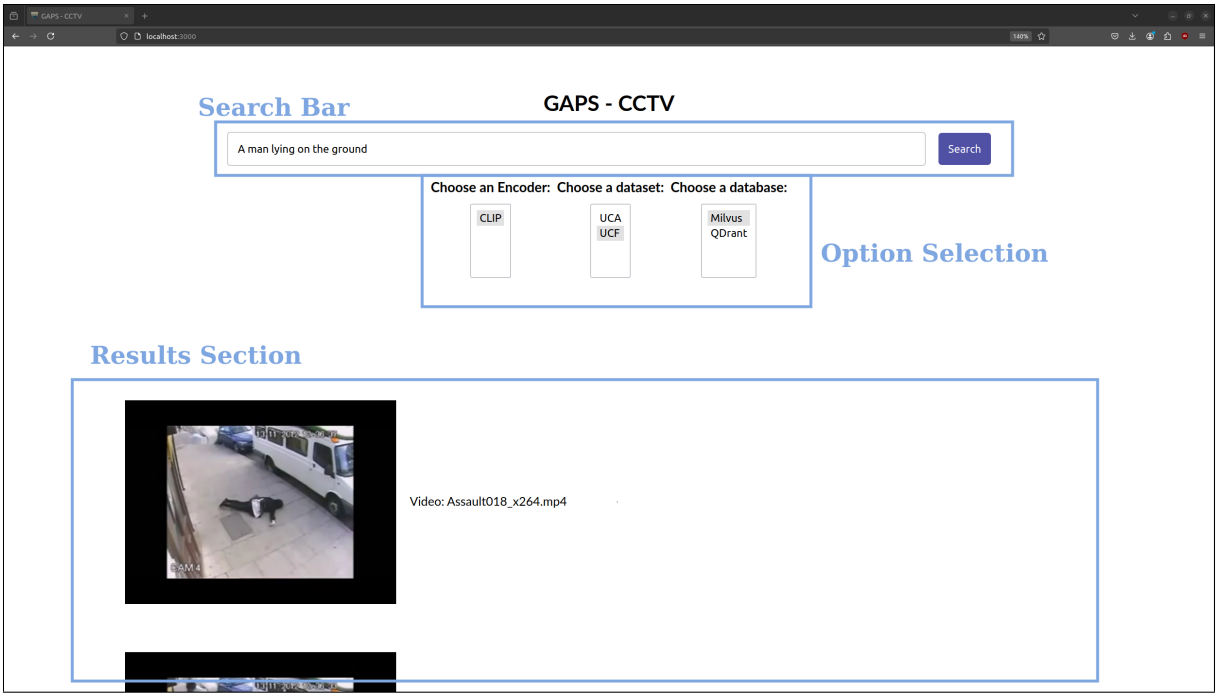

Figure 3.16: Components of the presentation.

- Database: refers to which database system to query, Milvus or Qdrant in our tests.

Obviously, these option menus are intended for debugging and testing purposes. A real application would hide such details, presenting to the user just one combination of encoder, dataset, and database.

Thus, the entire retrieval procedure performed by the user on the browser APP would then include the following steps:

1. Write a text prompt in the search bar.
2. Select options (encoder-database-dataset).
3. Click the search button and wait for the results.
4. Check the resulting video summaries.

The clicking of the search button triggers the execution of a Javascript code that fetches the URIs provided by the Business Layer REST API. The steps followed by the Javascript code are the following:

1. Validate the user request by assuring a prompt has been introduced while all the options have a valid selection.
2. Construct the URI of the form:

```
http://videosumHost:videosumPort/query/database?
                                          text=prompt&
                                    encoder=encoder&
                                     dataset=dataset
```

The terms in bold are the options and the prompt specified by the user.

3. Fetch the URI and define a callback to be executed when the response is obtained.
4. When the response is obtained:
    1. Get the JSON data transported in the response with all the information about the video summaries: video file name, start frame and end frame.
    2. Get all the videos from the Video Summarizer Server using `GET` requests to the URI:
    ```
    http://videosumHost:videosumPort/video?
                        name=video_file_name&
                    startFrame=start_frame&
                        endFrame=end_frame
    ```
    3. Display all video results (i.e., summaries) on the web browser.

3.6 Data flow

Now that all the components of the data retrieval application have been properly presented, a communication diagram for the flow of the application is shown in Figure 3.17.

Several things must be noted about this diagram:

- The client performs three different calls to the Video Summarization Server: one for obtaining the static content, another to get the video summaries information related to some text prompt, and a final one to get the fetch summaries.
- The Video Summary Server queries the vector database, and depending on the chosen dataset, we should expect a different answer:
    - If UCF: the key-frames information and has to build the video summary information from these frames.
    - If UCA: the video summaries information.

  Either way, the response to the client is the video summary information.
- The individual video summaries are generated and streamed from the Video Summarization Server, with each video summary streamed after an independent video request.

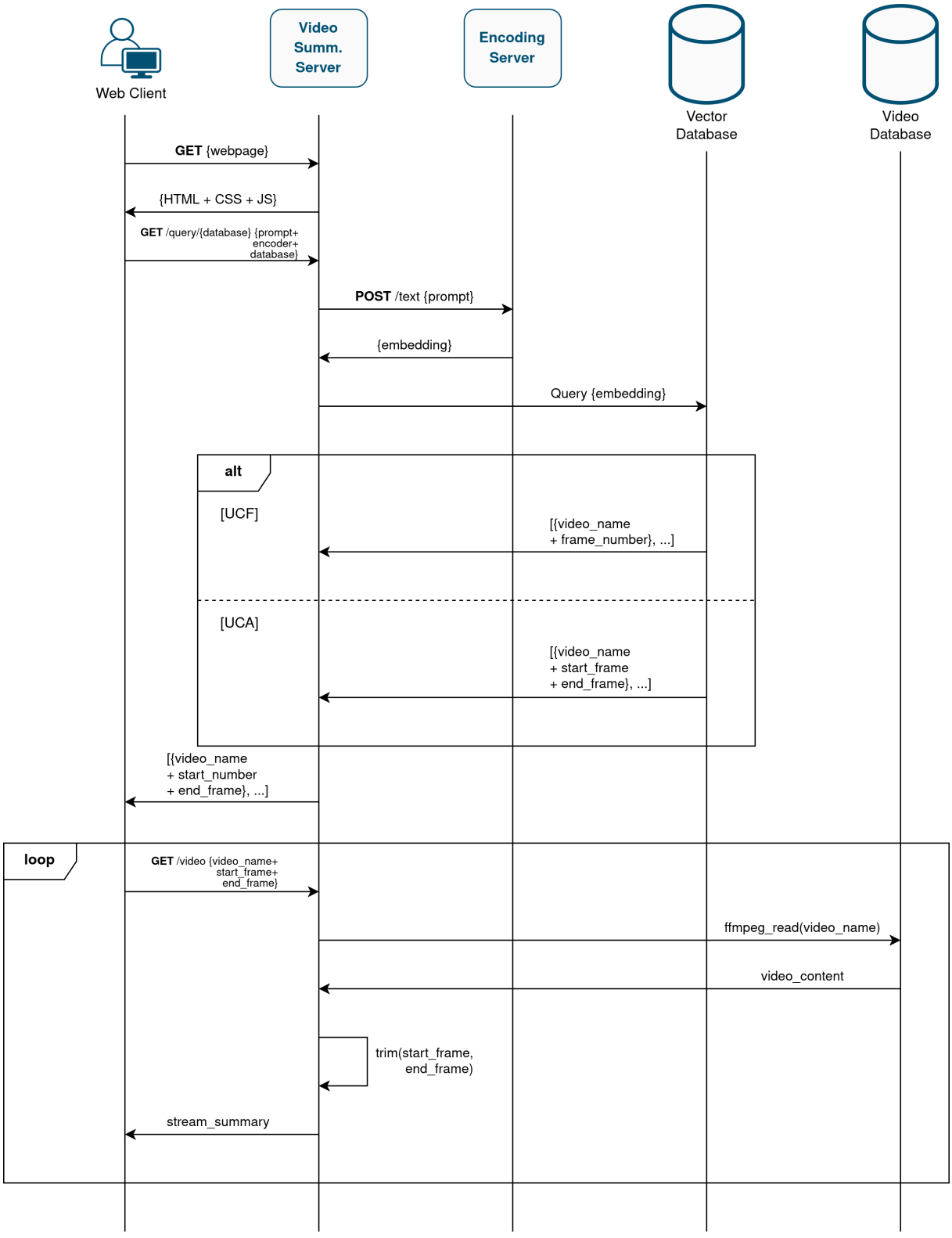

Figure 3.17: Full communication diagram of the retrieval application.

Chapter 4

Results

This chapter discusses the results obtained in this Master's Thesis. First, Section 4.1 evaluates the retrieval system using accuracy, availability and latency metrics. Then, Section 4.2 describes the final application with special emphasis on its user interface.

4.1 Metrics

This section describes the measurements taken to evaluate accuracy, availability and latency. A proper definition of these metrics is given at the beginning of each subsection.

4.1.1 Accuracy

We subjectively evaluate if the retrieved elements are relevant to the user or not attending to the input text query. Among the objectives of this work, we did not intend to develop the most accurate system possible. We just built a system that used cross-modal models (implementing CLIP as an example, but any other cross-modal model could be used) to get the summaries. However, no training or improvement over this model was tackled. We mainly tested the zero-shot capabilities of CLIP with our datasets. Nevertheless, the indexes of the databases can also impact the accuracy because their use reduces the recall (the resulting vectors may not be the most similar vectors in the dataset), so the measure of the accuracy is relevant.

To compute the values for this metric, we took several short sentences and evaluated whether the results were related to that sentence or not. To have a diverse set of test prompts, we defined four categories, each of them having four sentences, totalling 16 test prompts:

1. **General situations**: descriptions of typical actions or situations that do not describe any suspicious activity. These sentences use an individual or collective subject, a verb and some descriptive clauses.
    1.1. *A group of people talking outside.*
    1.2. *A dog running around.*

1.3. *A car driving through a highway.*

1.4. *A person buying in a shop.*

2. **Anomalous actions**: descriptions of rare, dangerous or suspicious situations related to UCF's categories. These sentences use an individual or collective subject, a verb and some descriptive clauses.

   2.1. *A fire burning in a house.*

   2.2. *People talking with anger.*

   2.3. *A car crashing in a street.*

   2.4. *A man lying on the ground.*

3. **Environment description**: a scene description, without mentioning any action or event, just the static elements. These sentences use a noun and some adjectives or descriptive clauses.

   3.1. *A rainy street.*

   3.2. *A parking lot at night.*

   3.3. *A crowded office.*

   3.4. *A forest in a sunny day.*

4. **Abstract text**: a text that uses abstract concepts to describe the scene. These sentences try to be generic and short, so only a noun with an adjective or a short descriptive clause is used.

   4.1. *A difficult situation.*

   4.2. *A happy place.*

   4.3. *Someone in danger.*

   4.4. *A tense moment.*

The scoring system for the results was established as follows:

- **4** points if all the resulting videos can be related to the text prompt.
- **3** points if most of the resulting videos can be related to the text prompt.
- **2** points if most of the resulting videos cannot be related to the text prompt.
- **1** points if none of the resulting videos can be related to the text prompt.

To decide whether a video summary is related or not with the text prompt, the tester took a subjective approach, but following some guidelines. For categories **1** and **2**, the described action or situation must be clearly shown in the video. For example, for sentence **2.2**, *People talking with anger*, if the video only showed people talking quietly or friendly, it would not have been considered relevant, since it was specified to show clear signs of anger. For category **3**, a video was considered relevant if it showed the exact environment described in the sentence, regardless of any of the actions taking place in the video.

For example, for sentence **3.2**, *A parking lot at night*, if the video showed a parking lot during daytime, it would not have been considered relevant. Finally, for category **4**, a less conservative approach was taken, taking a video as relevant if it showed something that could be minimally related to the text. For example, for sentence **4.2**, *A happy place*, if the video showed elements relatable to happy environments such as balloons, it would have been considered relevant.

Table 4.1 summarizes the evaluation results of the tests conducted by the author of this Master's Thesis. The first column specifies which sentence is being evaluated on each row. The next four columns specify the score obtained by each dataset-database combination. The next column specify the arithmetic mean of the 4 previous columns, showing the average score that sentence obtained for all the possible combinations. The final column specify the mean obtained from averaging all the means belonging to the category. This last column gives an overall performance indicator for each of the sentences' domain. Finally, the last row at the bottom of the table shows the mean of all the scores obtained using a specific dataset-database combination, showing its average performance.

Table 4.1: Results on the accuracy measure.

	UCF		**UCA**		**Mean**	**Category Mean**
	Milvus	**Qdrant**	**Milvus**	**Qdrant**		
1.1	4	4	3	3	**3.5**	**2.87**
1.2	1	1	1	1	**1**	
1.3	3	3	3	3	**3**	
1.4	4	4	4	4	**4**	
2.1	3	3	3	3	**3**	**2.31**
2.2	2	2	2	2	**2**	
2.3	1	1	2	2	**1.5**	
2.4	4	1	3	3	**2.75**	
3.1	4	4	1	3	**3**	**3.75**
3.2	4	4	4	4	**4**	
3.3	4	4	4	4	**4**	
3.4	4	4	4	4	**4**	
4.1	3	4	3	3	**3.25**	**3.31**
4.2	4	4	4	3	**3.75**	
4.3	4	4	3	3	**3.5**	
4.4	4	1	3	3	**2.75**	
Mean	**3.31**	**3**	**2.94**	**3**		

Regarding database-dataset combinations, the accuracy is pretty similar: Milvus-UCF scored the highest, and Milvus-UCA scored the lowest. Regarding categories, the retrieval system looks pretty accurate when the text prompt is a more straightforward environment description.

4.1.2 Availability

For availability, we computed the time it takes from the video footage recording to the moment it can be used for the summary generation. The availability evaluation was only

performed over the UCF dataset, and the UCA dataset was not considered. This is because UCA does not represent a realistic dataset for a CCTV system since the annotation process was done beforehand. If we had measured the availability of UCA, those values would have been meaningless since the process of annotating the videos is omitted. Thus, the three operations that must be evaluated are video encoding, vector ingestion and index creation.

For the video encoding, time measurements were computed directly from the encoding script. The encoding was done in indivisible batches of 64 frames, so we measured the time it took for each of these batches to be encoded into an embedding. Table 4.2 shows a statistical summary of the results. With the available hardware, it took 1.5 seconds to encode one frame on average. The standard deviation of the encoding time is substantial.

One of the main conclusions driven from Table 4.2 is related to the total time it took to encode the entire UCF-Crime dataset. The encoded video footage corresponded to around 113 hours of video[1], and the total encoding time is very close to that value. This fact demonstrates that a real-time encoding of the video as it is generated is feasible with the appropriate hardware.

Table 4.2: Statistics on the encoding time.

	UCF-Batch64 Encoding
Count (batches)	334,618
Mean (ms/batch)	1226.04
Standard deviation (ms/batch)	614.00
Min. (ms/batch)	161.22
Max. (ms/batch)	5904.04
25th Per. (ms/batch)	696.51
50th Per. (ms/batch)	1235.73
75th Per. (ms/batch)	1630.07
Total time	113.96 h

For the vector ingestion, we measured the time it took to ingest each embedding into the database. Table 4.3 summarizes the results. We can see that Milvus outperforms Qdrant regarding ingestion, and Qdrant presents a more significant variance for this task.

Index creation is the last process that must take place before the data can be retrieved. Qdrant builds its index automatically, so no additional time measurements can be taken. For Milvus, an IVF-PQ index was built, and it took 1.08 hours to build.

4.1.3 Latency

Latency is a metric related to the time the system takes since the text prompt is sent at the moment the video summaries are released. This metric can be divided into smaller chunks of time. We used a testing script to query each database (Milvus and Qdrant) 100 times with different randomly generated sentences.

[1]Hours of encoded video $= \frac{\text{Number of encoded frames}}{\text{FPS}\cdot 60^2} = \frac{12{,}247{,}318}{30\cdot 60^2} \approx 113.40$ hours

Table 4.3: Statistics on to the ingestion time.

	Milvus Ingestion	**Qdrant Ingestion**
Count (frames)	12,247,318	12,247,318
Mean (ms/frame)	2.75	4.93
Standard deviation (ms/frame)	10.26	31.51
Min. (ms/frame)	2.3	1.29
Max. (ms/frame)	15362.64	20967.96
25th Per. (ms/frame)	2.62	2.29
50th Per. (ms/frame)	2.68	3.85
75th Per. (ms/frame)	2.76	5.21
Total time	9.35 h	16.77 h

Next, we list the times measurements taken and the component which measured them:

- Prompt Encoding, measured by Video Summarization Server. The time passed from the moment the server sends the `POST` request to the Encoding Server until it receives the resulting embedding.
- Database Query, measured by Video Summarization Server. The time passed from the moment the server sends the query to the vector database until it receives the response.
- Video Trimming, measured by Video Summarization Server. The time it takes the server to trim the videos stored in the Video Database.
- Fetch Prompt, measured by the Client running the test script: the time passed from the moment it sends the `GET /query` request to the server response.
- Fetch Summary, measured by the Client running the test script: the time passed from the moment it sends the `GET /video` request to the server response.

These intervals can be properly identified in Figure 3.17.

4.1.3.1 UCF-Crime dataset

Table 4.4 summarizes some statistics on the measurements done in the Video Summarization Server during the execution of the test script. From all the operations performed by this server, the clear bottleneck is the Prompt Encoding. Another interesting conclusion is that Qdrant outperforms Milvus in query time, although it is true that Milvus query times present less variance than its counterpart. These outcomes seem to contradict the results of ingestion time in Table 4.3. Video Trimming is the quickest operation the Video Summarizer performs.

Table 4.5 summarizes the statistics of the measurements on the Client side during the execution of the test script. From all the operations, fetching the videos requires less time than the others. It should be noted that the video fetching operation does not mean

Table 4.4: Statistics on the latency measurements for the Video Summarization Server using UCF-Crimes dataset.

	Prompt Encoding	**Database Query (Milvus)**	**Database Query (Qdrant)**	**Video Trimming**
Count (requests)	200	100	100	2000
Mean (ms/request)	3100.10	545.87	294.17	242.73
Standard deviation (ms/request)	153.35	32.90	146.86	72.39
Min. (ms/request)	2939.00	504.00	4.00	83.00
Max. (ms/request)	3754.00	761.00	857.00	529.00
25th Per. (ms/request)	3002.00	527.75	223.00	188.00
50th Per. (ms/request)	3038.00	538.50	298.00	239.50
75th Per. (ms/request)	3138.25	554.00	377.00	282.25
Total time	10.42 min	0.92 min	0.49 min	8.20 min

downloading the entire video; the operation is considered completed when the server starts streaming the video. As for Milvus and Qdrant fetching, we can see a very close set of results. The comparison between databases yields the same conclusions as in Table 4.4. Qdrant is quicker than Milvus, but Milvus presents less variance than Qdrant.

Table 4.5: Statistics on the latency measurements of the Client using UCF-Crimes dataset.

	Fetch Prompt (Milvus)	**Fetch Prompt (Qdrant)**	**Fetch Summary**
Count (requests)	100	100	2000
Mean (ms/request)	3661.93	3385.90	278.38
Standard deviation (ms/request)	168.28	203.79	78.85
Min. (ms/request)	3475.02	3013.39	112.02
Max. (ms/request)	4219.57	4175.38	597.36
25th Per. (ms/request)	3547.60	3274.04	218.99
50th Per. (ms/request)	3594.05	3352.40	272.14
75th Per. (ms/request)	3728.11	3481.40	323.12
Total time	6.10 min	5.64 min	9.28 min

4.1.3.2 UCA dataset

Table 4.6 shows a statistical summary of the measurements done in the Video Summarization Server during the execution of the test script, this time using UCA dataset. As expected, the Prompt Encoding times are similar to those shown in Table 4.4, the total encoding time in this case being only a minute longer. Nonetheless, there are interesting results regarding the database query times. While Milvus presents a similar performance compared to querying UCF dataset, Qdrant improves its performance by resolving the queries in a matter of milliseconds, which is astonishing. The superiority of Qdrant com-

pared to Milvus becomes even more apparent since, for the UCA dataset, both implement a HNSW index.

Finally, we can see a variance increase in the Video Trimming times. This outcome is because, in the case of UCF dataset, the trimming operation always cuts 5-second chunks, while in the case of UCA, the trimming operation cuts chunks with different lengths.

Table 4.6: Statistics on the latency measurements of the Video Summarization Server using UCA dataset.

	Prompt Encoding	**Database Query (Milvus)**	**Database Query (Qdrant)**	**Video Trimming**
Count (requests)	200	100	100	2000
Mean (ms/request)	3453.58	571.49	5.72	505.12
Standard deviation (ms/request)	316.47	166.50	2.08	458.74
Min. (ms/request)	2981.00	525.00	4.00	64.00
Max. (ms/request)	4284.00	2220.00	14.00	8073.00
25th Per. (ms/request)	3166.75	544.00	4.00	226.75
50th Per. (ms/request)	3433.00	552.50	5.00	380.50
75th Per. (ms/request)	3719.25	565.50	7.00	645.75
Total time	11.51 min	0.95 min	0.57 sec	16.84 min

Table 4.7 shows a statistical summary of the measurements done in the Client during the execution of the test script, this time using UCA dataset. From all the operations, fetching the videos is still the one which requires less time. As for Milvus and Qdrant fetching, we can see that the difference between mean values is coherent with the difference shown in Table 4.6, showing that Qdrant outperforms Milvus by far.

Table 4.7: Statistics on the latency measurements of the Client using UCA dataset.

	Fetch Prompt (Milvus)	**Fetch Prompt (Qdrant)**	**Fetch Summary**
Count (requests)	100	100	2000
Mean (ms/request)	4034.25	3457.75	606.95
Standard deviation (ms/request)	341.38	330.42	530.02
Min. (ms/request)	3538.58	2988.58	96.55
Max. (ms/request)	5404.70	4296.13	6633.38
25th Per. (ms/request)	3729.47	3173.43	264.06
50th Per. (ms/request)	4073.47	3400.08	452.56
75th Per. (ms)	4250.01	3755.95	764.26
Total time	6.72 min	5.76 min	20.23 min

4.2 Application

The primary outcome of this project is the web application. For the application to be accessed, we must go to the browser and look for URI `http://<nodejs-ip>:3000/`. When the page is loaded, the user gets a similar figure as the one Figure 4.1. The app offers an intuitive interface so the user can interact with the elements easily, without previous training to operate the app. This intuitive design was achieved thanks to the imperative sentences («Introduce a search text...», «Choose a dataset», etc.).

The first thing the user sees is the title «GAPS - CCTV», providing the context of the app: it was developed inside the GAPS research group and is used for CCTV systems. Next, the user can find the search bar, where the text prompt is introduced. Next to the bar is the search button, which triggers the retrieval procedure, although pressing the «Enter» key also works. Under this search bar, the set of options for the retrieval can be selected. The bottom part shows the *Universidad Politécnica de Madrid* logo and the *Grupo de Aplicaciones del Procesado de Señal* logo.

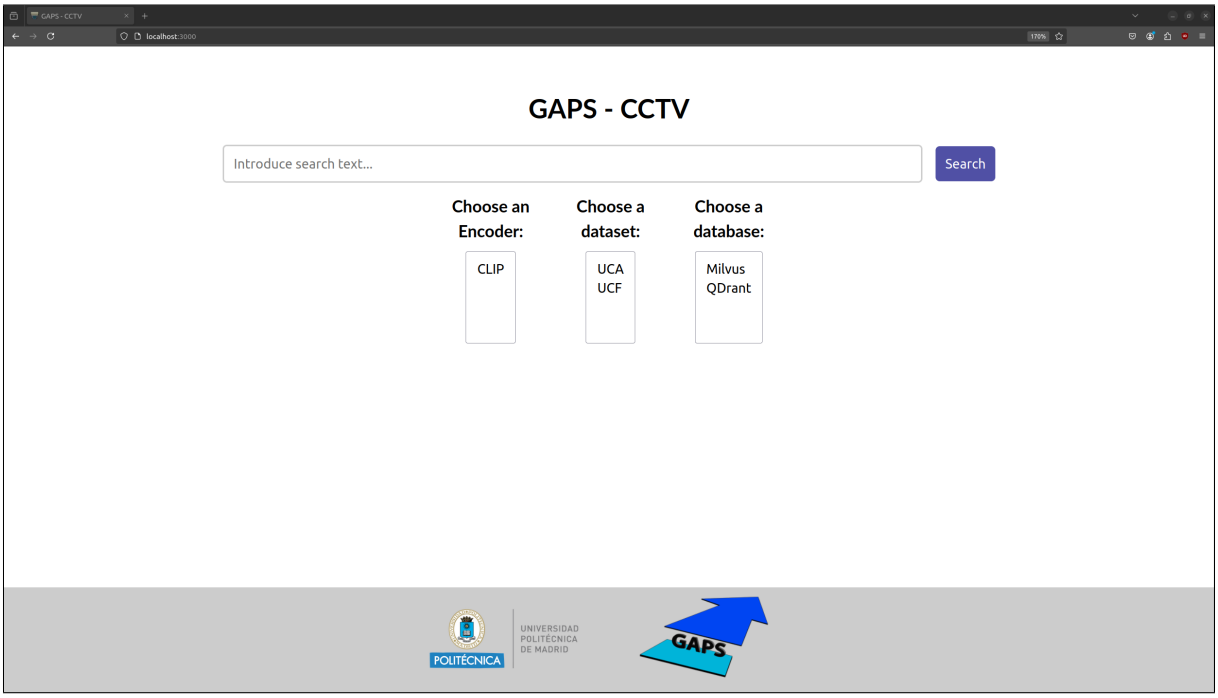

Figure 4.1: Initial view of the application.

Figure 4.2 shows the introduction of a prompt in the search bar.

Figure 4.3 shows the option selection. In this case, CLIP encoder, UCF dataset and Qdrant database were selected.

Both the procedures shown in Figures 4.2 and 4.3 must be done before the «Search» button or the «Enter» key is pressed. If the user does not introduce a text in the text prompt, the error message shown in Figure 4.4 appears. If the prompt has been introduced but any of the options but the user has not chosen an option, a message like the one shown in Figure 4.5 appears, stating which of the options is not selected.

Finally, when the prompt has been introduced and all the options are selected, the user can press the «Search» button. After a short time, a list of summary videos is presented

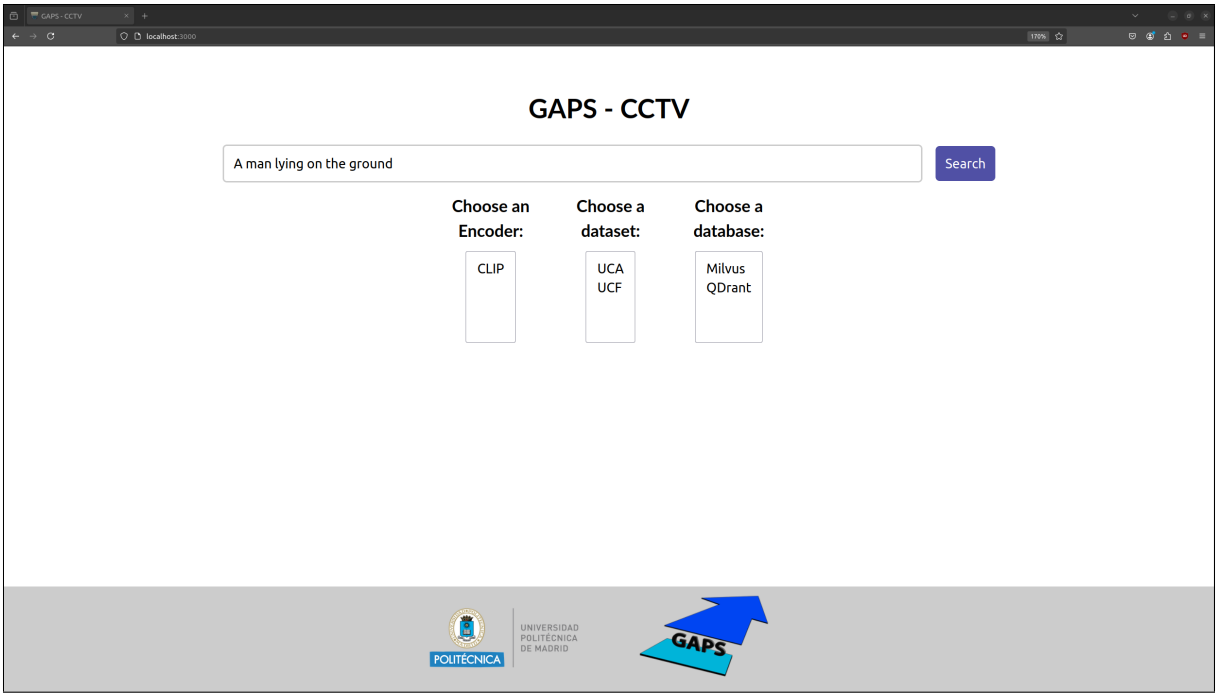

Figure 4.2: Prompt introduction in the search bar.

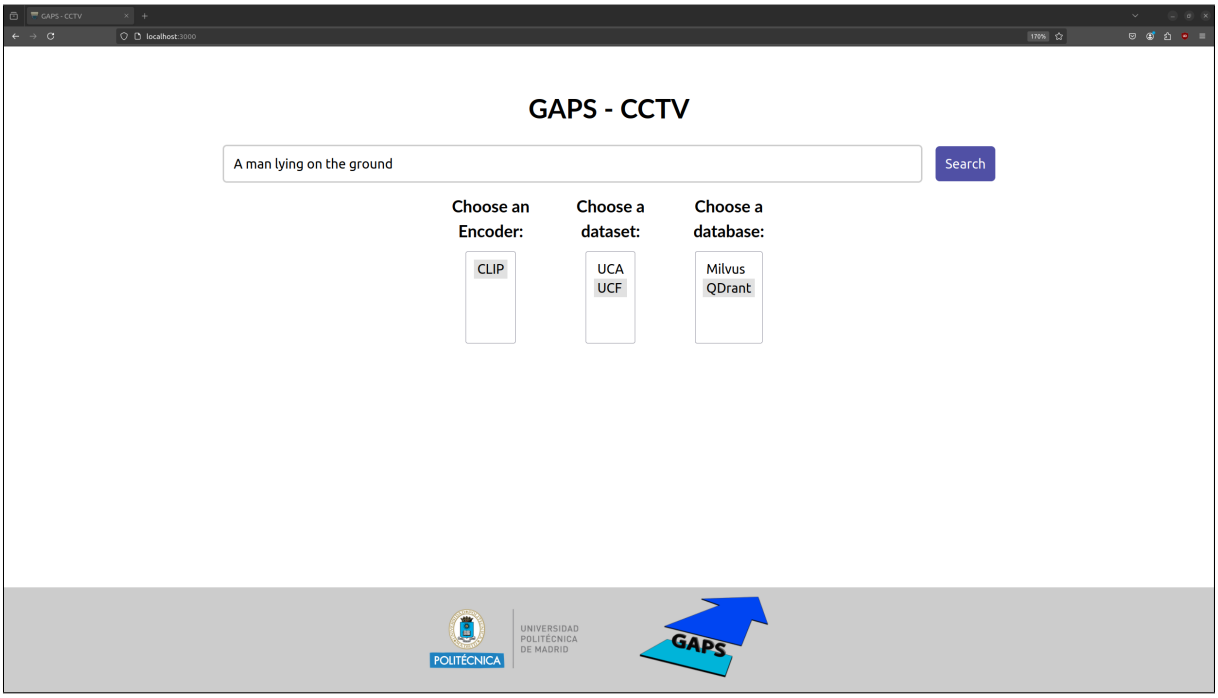

Figure 4.3: Options selection.

on the screen, as shown in Figure 4.6. The user can press the play button any of the videos and visualize its content. Next to each video, we include the name of the original source video file.

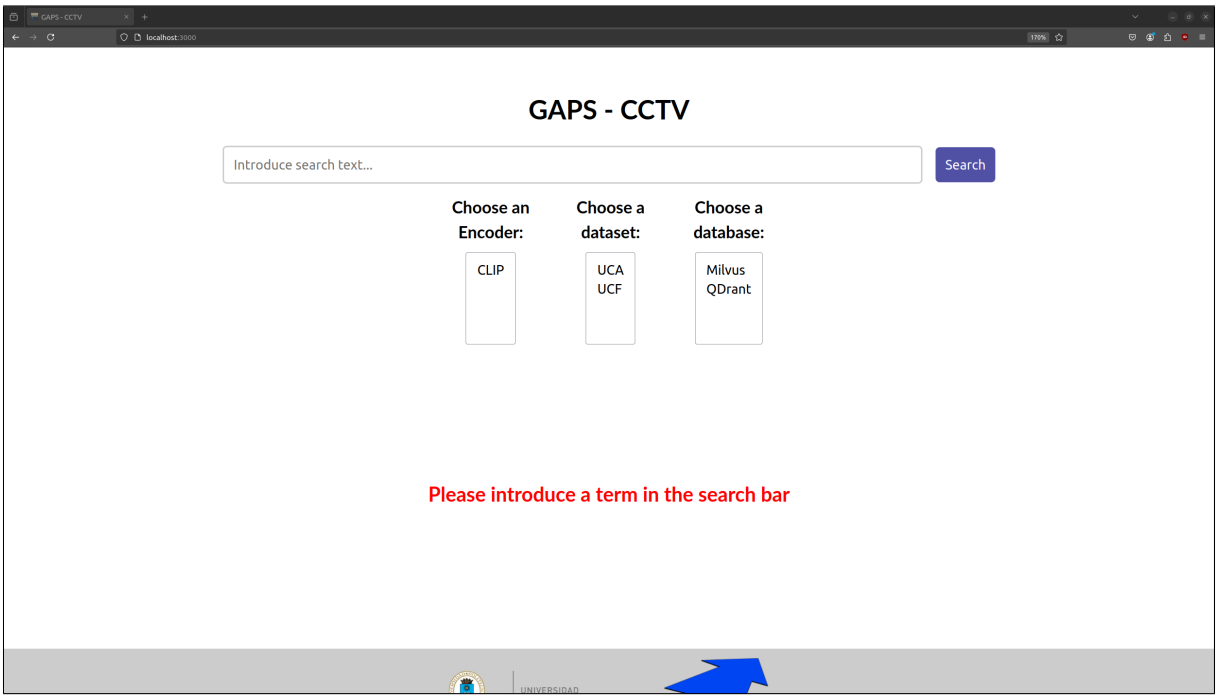

Figure 4.4: Error shown when prompt has not been introduced.

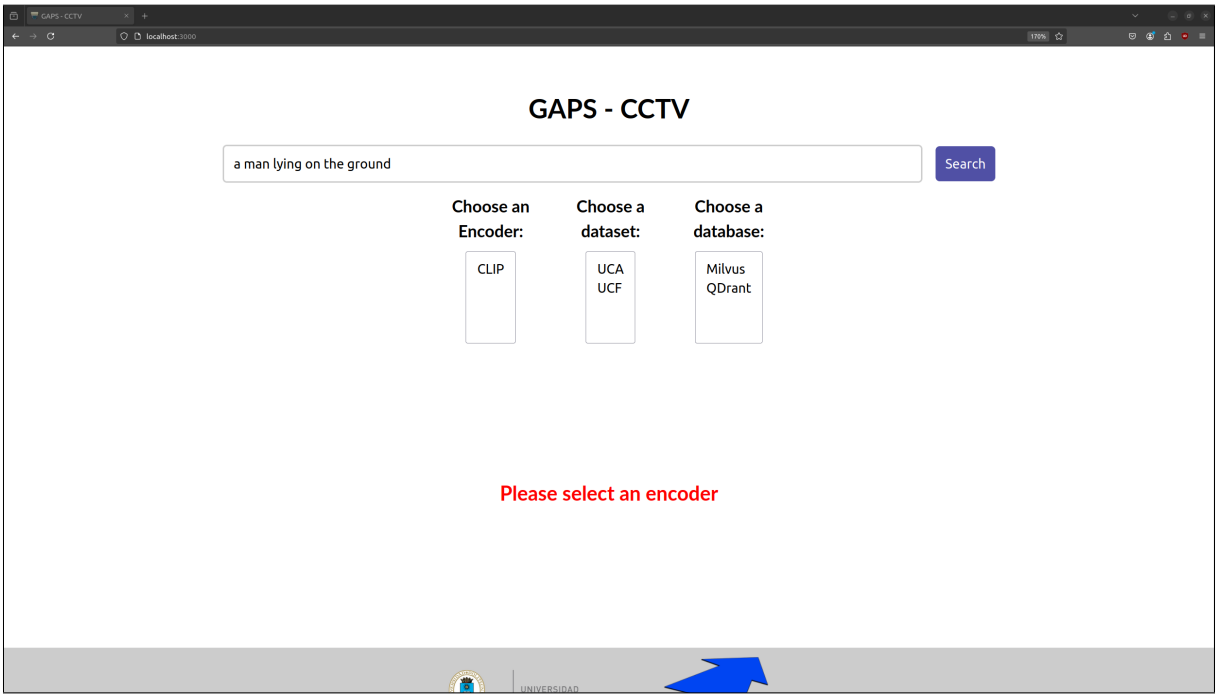

Figure 4.5: Error shown when the encoder has not been introduced.

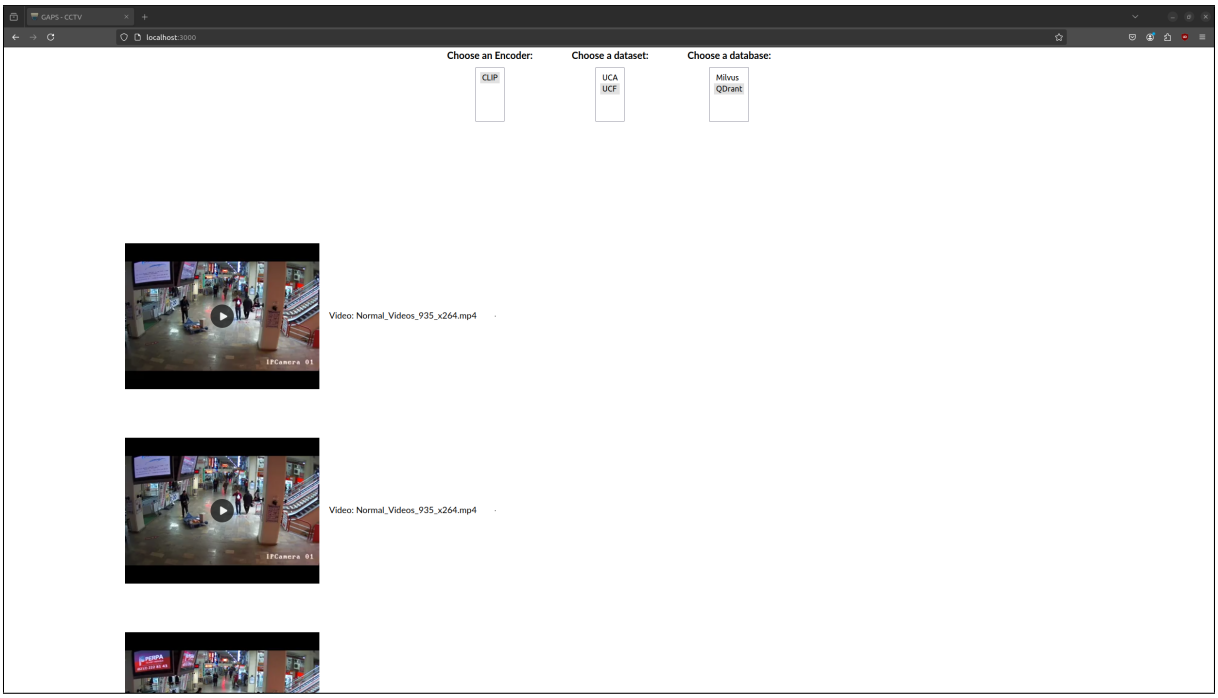

Figure 4.6: Streaming of summary videos.

Chapter 5

Conclusion

This chapter summarizes our conclusions regarding the outcomes of this work, paying special attention to the objectives stated in Chapter 1. We include a series of future directions to improve and extend this work, distinguishing between short-term and long-term proposals.

5.1 Conclusion

This Master's Thesis has developed a video retrieval and summarization application for a video surveillance system using cross-modal vector representations of multimedia data. We have demonstrated the great potential of cross-modal embeddings in a common multimedia-intensive application, leveraging data ingestion and information retrieval capabilities.

The first stage of this project involved extensive research on cross-modal embedding technology, which demonstrated that vector representations are valuable tools for building retrieval and summarization systems involving multimedia assets. They exhibit an inherent ability to encapsulate the semantic properties of multimedia data supported by a simple similarity measure.

The preliminary research stage covered vector databases and vector management systems, the most suitable tools for large-scale applications. They successfully leverage the properties of cross-modal representations while providing the speed and scalability we require. This kind of database is recent but presents a promising future in the industry. Milvus and Qdrant were the databases selected to be deployed in this application, representing a suitable option for a CCTV system.

Having finished the initial research, we tackled the implementation of the retrieval application, which was this project's main objective. As presented in Chapter 3, this was done through the development of several independent components which communicate between them to provide the desired functionality, all under the architectural framework of a layered application.

This work demonstrated the viability of building a retrieval and summarization application through cross-modal AI technology for CCTV systems. The final application is fully functional and can adapt to different CCTV systems. Due to the great decoupling of each

component and the abstraction provided by the APIs, they can be adapted with a few changes to any other system requiring a summarization and retrieval application. However, the final solution did not include real-time ingestion capabilities but purely focused on the retrieval.

Regarding the secondary goals listed in Chapter 1, we reach the following conclusions.

1. *Familiarization with cross-modal representations or embeddings*: all this project has pivoted around this kind of data. First, in Section 2.3, we introduced the concept of cross-modal representations obtained from DL models. Section 3.3.2 described how this data can be obtained from a large video dataset using several approaches. Finally, Section 3.4.2 showed how text representations can be used to query a database of cross-modal representations to retrieve images similar to a given text.

2. *Familiarization with vector database systems and comparative study of different implementations*: We made an extensive analysis of vector database management systems in Section 2.4, where we discussed about these systems, their use and how and what implementations are available in the market; comparing the most representative of them in Table 2.1. Additionally, we explained how vector indexes can ease the data query process. Then, throughout Section 3.3.2, we described two specific VDBMS implementations, Milvus and Qdrant, stating their differences. Finally, in Sections 4.1.2 and 4.1.3, we studied the performance of ingestion and query operations, respectively, for each database and compared the results.

3. *Deployment of index structures to search vector databases*: Section 3.3.2.2 included an explanation of how different indexes can be built in Milvus and the criteria used to select the index type and its parameters.

4. *Formulation and completion of data ingestion benchmarks*: Section 4.1.2 shows and analyses the results obtained from the ingestion process.

5. *Formulation and completion of data retrieval benchmarks*: Section 4.1.3 shows and analyses the results obtained from the retrieval process.

In summary, all goals have been successfully covered, proving that the proposed scheme could be used for video retrieval, integrating within a retrieval pipeline that could use any of the two most common vector databases.

5.2 Future directions

We consider two paths to expand on this work in the short and the long terms. For the short-term lines of improvement, we propose the following:

- Use actual surveillance camera footage and test the system with a more realistic CCTV data: this could not be done in this work due to lack of resources, but it is the immediate step forward. It requires both actual CCTV data and the infrastructure to ingest, store effectively, and index all these data. We have requested a new dataset from a public agency that could be used for this purpose. Such validation must be addressed as a pilot for a future project.

- Deploy this system in a real CCTV environment: This work has tried to implement this system so it can be deployed along other real CCTV components. A full deployment inside the context of a surveillance system is necessary to test the performance of these tools.
- Deploy techniques to increase the diversity of the results: a problem that has been detected during the testing phase of the development is that, for some prompts, the results were frames too close in the video footage, especially when using UCF dataset. Having two or more similar summaries cannot be considered inaccurate; this irrelevant data could hide important content, so this issue must be tackled.

For the long-term, lines for improvement may include the following.

- Use of different modalities: This work has only considered text as the input prompt and images as the search asset, but other modalities could be used, such as image as prompt or audio as retrieval asset, or even combining more than one modality at once. This approach can increase the summaries' accuracy and cope with a modality's limitations. We suggest two steps to implement this advancement: one is to study other models that can combine different data types, such as CLAP, a contrastive pair of transformers for audio and text data. The other is to leverage the multi-vector query capabilities of VDBMS and explore multi-vector storage for a single asset.
- Develop better indexing algorithms: Indexing is a key step in obtaining accurate results the fastest way possible. The development of vector-data indexes is a field of study showing new advancements each day, but there is still room for improvement. Developing an index suitable for the kind of vectors this system works with is a key future line for improving digital assets management systems in general. In particular, a combination of indexes could be used to condense the embedded information effectively for video assets. Graph structures are among the most effective and may be combined into rich knowledge representations.

References

[1] *The volume of the global smart video surveillance market for the year grew to $12.28 billion — tadviser.com*, https://tadviser.com/index.php/Article:Video_Surveillance_(Global_Market), [Accessed 28-01-2025].

[2] *626 million surveillance cameras are all over china: 432.2 surveillance cameras per 1,000 people — chinascope.org*, https://chinascope.org/archives/30749, [Accessed 28-01-2025].

[3] C. Shi and J. Xu, "Surveillance cameras and resistance: A case study of a middle school in china," *The British Journal of Criminology*, vol. 64, no. 5, pp. 1150–1170, Jan. 2024, ISSN: 0007-0955. DOI: 10.1093/bjc/azad078. eprint: https://academic.oup.com/bjc/article-pdf/64/5/1150/58890726/azad078.pdf. [Online]. Available: https://doi.org/10.1093/bjc/azad078.

[4] O. Elharrouss, N. Almaadeed, and S. Al-Maadeed, "A review of video surveillance systems," *Journal of Visual Communication and Image Representation*, vol. 77, p. 103 116, 2021, ISSN: 1047-3203. DOI: https://doi.org/10.1016/j.jvcir.2021.103116. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1047320321000729.

[5] H. Kruegle and F. Abram, "Chapter 1 - video's critical role in the security plan," in *CCTV Surveillance (Second Edition)*, H. Kruegle and F. Abram, Eds., Second Edition, Burlington: Butterworth-Heinemann, 2006, pp. 1–11, ISBN: 978-0-7506-7768-4. DOI: https://doi.org/10.1016/B978-075067768-4/50004-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780750677684500046.

[6] F. Porikli, F. Bremond, S. L. Dockstader, *et al.*, "Video surveillance: Past, present, and now the future [dsp forum]," *IEEE Signal Processing Magazine*, vol. 30, no. 3, pp. 190–198, 2013. DOI: 10.1109/MSP.2013.2241312.

[7] *Cisco video surveillance solution; reference network design guide*, Cisco Systems Inc., 2013. [Online]. Available: https://www.cisco.com/c/dam/en/us/td/docs/security/physical_security/video_surveillance/network/vsm/7_0/srnd/vsm_7_0_srnd.pdf.

[8] P. Meena, H. Kumar, and S. Kumar Yadav, "A review on video summarization techniques," *Engineering Applications of Artificial Intelligence*, vol. 118, p. 105 667, 2023, ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2022.105667. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197622006571.

[9] C. Cuevas, C. R. del Blanco, N. García, and J. Cabrera, *Lecture notes in vision analysis and deep learning*, 2023.

[10] C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*. Springer, 2024.

[11] A. Dosovitskiy, L. Beyer, A. Kolesnikov, *et al.*, *An image is worth 16x16 words: Transformers for image recognition at scale*, 2021. arXiv: 2010.11929 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2010.11929.

[12] A. Arnab, M. Dehghani, G. Heigold, C. Sun, M. Lučić, and C. Schmid, *Vivit: A video vision transformer*, 2021. arXiv: 2103.15691 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2103.15691.

[13] A. Radford, J. W. Kim, C. Hallacy, *et al.*, *Learning transferable visual models from natural language supervision*, 2021. arXiv: 2103.00020 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2103.00020.

[14] kingyiusuen, *Image search using clip*. [Online]. Available: https://github.com/kingyiusuen/clip-image-search.

[15] B. Dudhrejiya, *Content moderation system using clip*, https://medium.com/@bhavik.datascience/content-moderation-system-using-clip-bd1ba1896c28, Accessed: 2025, 2023.

[16] R. Mokady, A. Hertz, and A. H. Bermano, *Clipcap: Clip prefix for image captioning*, 2021. arXiv: 2111.09734 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2111.09734.

[17] I. G. S. M. Diyasa, A. D. Alhajir, A. M. Hakim, and M. F. Rohman, "Reverse image search analysis based on pre-trained convolutional neural network model," in *2020 6th Information Technology International Seminar (ITIS)*, 2020, pp. 1–6. DOI: 10.1109/ITIS50118.2020.9321037.

[18] M. Grbovic and H. Cheng, "Real-time personalization using embeddings for search ranking at airbnb," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18, London, United Kingdom: Association for Computing Machinery, 2018, pp. 311–320, ISBN: 9781450355520. DOI: 10.1145/3219819.3219885. [Online]. Available: https://doi.org/10.1145/3219819.3219885.

[19] T. Taipalus, "Vector database management systems: Fundamental concepts, use-cases, and current challenges," *Cognitive Systems Research*, vol. 85, p. 101 216, Jun. 2024, ISSN: 1389-0417. DOI: 10.1016/j.cogsys.2024.101216. [Online]. Available: http://dx.doi.org/10.1016/j.cogsys.2024.101216.

[20] M. Douze, A. Guzhva, C. Deng, *et al.*, *The faiss library*, 2024. arXiv: 2401.08281 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2401.08281.

[21] S. Kukreja, T. Kumar, V. Bharate, A. Purohit, A. Dasgupta, and D. Guha, "Vector databases and vector embeddings-review," in *2023 International Workshop on Artificial Intelligence and Image Processing (IWAIIP)*, 2023, pp. 231–236. DOI: 10.1109/IWAIIP58158.2023.10462847.

[22] K. Aoyama, K. Saito, and T. Ikeda, *Inverted-file k-means clustering: Performance analysis*, 2020. arXiv: 2002.09094 [stat.ML]. [Online]. Available: https://arxiv.org/abs/2002.09094.

[23] Y. A. Malkov and D. A. Yashunin, *Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs*, 2018. arXiv: 1603.09320 [cs.DS]. [Online]. Available: https://arxiv.org/abs/1603.09320.

[24] E. Bernhardsson, *Annoy: Approximate nearest neighbors in c++/python*, 2018. [Online]. Available: https://pypi.org/project/annoy/.

[25] F. Liu, *Approximate Nearest Neighbors Oh Yeah (Annoy) - Zilliz Learn — zilliz.com*, https://zilliz.com/learn/approximate-nearest-neighbor-oh-yeah-ANNOY, [Accessed 05-01-2025], 2023.

[26] M. Aumüller, E. Bernhardsson, and A. Faithfull, *Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms*, 2018. arXiv: 1807.05614 [cs.IR]. [Online]. Available: https://arxiv.org/abs/1807.05614.

[27] D. Quarry, *Vector databases (4): Analyzing the trade-offs*, https://thedataquarry.com/posts/vector-db-4/, [Accessed 05-01-2025], 2023.

[28] N. Yuhanna, A. Katz, S. Sjoblom, and J. Barton, *The forrester wave™: Vector databases, q3 2024*, https://www.forrester.com/report/the-forrester-wave-tm-vector-databases-q3-2024/RES181372, Sep. 2024.

[29] "Vector database market size - by component, by technology, by industry vertical, analysis, share, growth forecast, 2025 - 2034," Global Market Insights, Tech. Rep., Dec. 2024.

[30] "Vector database market by offering (solutions and services), technology (nlp, computer vision, and recommendation systems), vertical (media & entertainment, it & ites, healthcare & life sciences) and region - global forecast to 2028," Markets and Markets, Tech. Rep., Oct. 2023.

[31] Andrew Kane, *Pgvector*. [Online]. Available: https://github.com/pgvector/pgvector.

[32] W. Yang, T. Li, G. Fang, and H. Wei, "Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2241–2253, ISBN: 9781450367356. DOI: 10.1145/3318464.3386131. [Online]. Available: https://doi.org/10.1145/3318464.3386131.

[33] Y. Zhang, S. Liu, and J. Wang, "Are there fundamental limitations in supporting vector data management in relational databases? a case study of postgresql," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 3640–3653. DOI: 10.1109/ICDE60146.2024.00280.

[34] R. Software, *DB-Engines Ranking — db-engines.com*, https://db-engines.com/en/ranking, [Accessed 09-01-2025], Jan. 2025.

[35] J. Gutiérrez Navarro, "Estudio para el desarrollo de un sistema de apoyo a la anotación de video con un modelo agnóstico de imagen y texto," Jul. 2023. [Online]. Available: https://oa.upm.es/75341/.

[36] *Papers with Code - UCF-Crime Dataset — paperswithcode.com*, https://paperswithcode.com/dataset/ucf-crime, [Accessed 17-01-2025].

[37] W. Sultani, C. Chen, and M. Shah, *Real-world anomaly detection in surveillance videos*, 2019. arXiv: 1801.04264 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1801.04264.

[38] J. Wang, X. Yi, R. Guo, *et al.*, "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21, Virtual Event, China: Association for Computing Machinery, 2021, pp. 2614–2627, ISBN: 9781450383431. DOI: 10.1145/3448016.3457550. [Online]. Available: https://doi.org/10.1145/3448016.3457550.

[39] *Qdrant - Vector Database — qdrant.tech*, https://qdrant.tech/, [Accessed 17-01-2025].

[40] Itseez, *Open source computer vision library*, https://github.com/itseez/opencv, 2015.

[41] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: http://dx.doi.org/10.1038/s41586-020-2649-2.

[42] *Milvus vector database documentation — milvus.io*, https://milvus.io/docs/v2.4.x, [Accessed 17-01-2025].

[43] *Choose the k-NN algorithm for your billion-scale use case with OpenSearch | Amazon Web Services — aws.amazon.com*, https://aws.amazon.com/blogs/big-data/choose-the-k-nn-algorithm-for-your-billion-scale-use-case-with-opensearch/, [Accessed 27-01-2025].

[44] P. G. D. Group, *PostgreSQL*, http://www.postgresql.org, 2008.

[45] *Node.js — Run JavaScript Everywhere — nodejs.org*, https://nodejs.org/, [Accessed 21-01-2025].

[46] *Flask Documentation (3.1.x) — flask.palletsprojects.com*, https://flask.palletsprojects.com/en/stable/, [Accessed 17-01-2025].

[47] K. Sharma, "Surveillance cameras in cities: A threat to privacy?" *Observer Research Foundation*, 2021.

[48] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, "A survey on bias and fairness in machine learning," *ACM Computing Surveys*, vol. 54, no. 6, pp. 1–35, 2021. DOI: 10.1145/3457607.

[49] O. Bohdal and S. Tolan, "Fairness in ai and its long-term implications on society," *arXiv preprint arXiv:2304.09826*, 2023.

Annexes

A Ethical, economic, social and environmental considerations

A.1 Introduction

In this Annex, a discussion of the Master Thesis' ethical, social, economic, and environmental impacts will be given. The general ponders of this section revolve around the impact of artificial intelligence tools in CCTV systems, and how the use of advance content retrieval and summarization techniques may affect public or private surveillance.

A.2 Description of relevant impacts related to the project

- **Ethical**: one positive impact of this work is the increase in safety. Fast retrieval systems and summarizations provide a shorter reaction time for security and emergency personnel. Also, automated systems may reduce human-induced biases in manually reviewed surveillance footage. Regarding the possible negative ethical impacts, privacy is a general concern around CCTV systems, and the application developed in this Master Thesis does not tackle this issue directly. Also, although it was said that automated systems may reduce human bias, the fact that the summaries generation depends on a humanly written prompt can induce biases in the application. The existence of model-level bias is not disposable either. Similarly, optimizations on the summarization tool from human prompts may condition the results obtained.
- **Economic**: the main economic impact of this work is the increase in cost efficiency due to the summarization process' reduction of time, video storage costs, and inefficient human labor. Also, it presents a tremendous economic opportunity since the system is built with standard components, favouring cost-effective scalability and shorter transition periods.
- **Social**: as stated in the ethical impact point, this application could enhance public and private safety and improve responses to possible dangers so personal and material damages can be substantially reduced. Also, a good use of this system may change the public perception of artificial intelligence by providing significant benefits to public users. Finally, this system can contribute to the development of smart cities and the improvement of the urban environment, which helps increase the quality of life in societies.

- **Environmental**: the use of vector data as multimedia content representation dramatically reduces computer usage when managing that kind of assets by allowing systems to perform operations related to the content without actually processing the multimedia content, which would be highly CPU-demanding. Additionally, using general-purpose computing hardware to deploy the components listed in this project contributes to reducing electric and electronic waste.

A.3 Detailed analysis of some of the main impacts

The primary and most immediate impact area of the work developed in this Master Thesis is ethical, with the application showing potentially positive and negative angles.

Among the positive impacts, there is an increase in safety due to the improved efficacy of CCTV systems. The key lies in reducing the reaction time, driven by two main factors: the ease of the information-gathering process after some event has occurred and the faster diagnosis of anomalous or alarming situations. Furthermore, the fact that the video summarization process is entirely automatic and systematic and can run autonomously, with minimal human intervention, has positive implications for privacy and data management concerns.

Also, automated systems can reduce humanly induced bias when reviewing video footage regarding race, gender, etc. The result is a fairer and safer surveillance system, fairer and safer for everyone, meditating discrimination against any individual arising from their usage. This is ensured as long as the model's training preserves such principles. The latter has been an open topic of debate in the past few years.

Among the ethical challenges of this application, privacy concerns are among the most critical. Since the summaries are generated depending on a user-introduced prompt, this application may be used for individual surveillance targeting. This is a concern for any CCTV system, but the fact that this application eases the CCTV general usage also means it eases the unethical CCTV usage. Preserving privacy is a critical topic for discussion [47]. One that requires considerable effort and continuous improvement.

Another issue is that, although automated systems reduce human bias, the prompts used to generate the summaries are not bias-protected and, therefore, can induce bias in the system [48].

Nevertheless, a series of strategies must be implemented to reduce the ethical harm this system may convey or at least to retain control on the operations of the system:

- Making extensive system audits helps ensure the application's fairness and safety. Master Data Management principles and correct Data Governance can provide the required privacy and secure our data.
- Deploying an effective access control in the system and ensuring only qualified and properly trained individuals can perform searches reduces human bias. Furthermore, an extensive analysis of the results may help improve the system and identify queries that mitigate such biases.
- Informing users of the existence and presence of this kind of surveillance system is an effective way of safeguarding individuals' privacy. It also allows them to exercise their

rights, which may require a mailbox or a similar tool to respond to users' demands.

- Using only contrasted and fair models in implementing this application is key to reducing biases. Nevertheless, the contrast and fairness of the models are still a matter of discussion amongst the scientific community [49].

A.4 Conclusions

Bringing cross-modal artificial intelligence to digital asset management has a considerable impact. Providing efficient procedures to build systems around unstructured multimedia data, such as retrieval or summarization applications, is a lengthy and costly process. This has an important economic impact on companies that deal with this kind of data, making them more cost-effective and improving key procedures.

For this specific application, CCTV systems benefit from automatic content retrieval and summarization by easing the reviewing work. These capabilities make them more effective in terms of safety. Also, a fair summarization system can help overcome any possible human bias, but the bias can be introduced through the input prompt.

To sum up, after studying the possible ethical, economic, social, and environmental impacts of this work and paying special attention to both ethical impacts, we believe the use of cross-modal representations-based content retrieval systems is a fair advancement in the world of digital assets management application. Our approach can bring fruitful benefits to companies and institutions, believing one can overcome this work's challenges and negative impacts with the proper mitigation strategies.

B Economic budget

STAFF COST	**Hours**	**Price/Hour**	**Total**
	780	55 €	**42,900 €**

MATERIAL RESOURCES COST	**Buy Cost**	**Month usage**	**Amortization (in years)**	**Total**
Storage and processing server	14,859.17€	6	5	1485.92€
GPUs (x2)	1019.98€	3	5	51€
GPUs equipment	1489.23€	3	5	74.46€

TOTAL COST OF MATERIAL RESOURCES	**1611.38€**

GENERAL EXPENDITURE	15%	over DC	**6676.71€**
INDUSTRIAL INCOME	6%	over DC+IC	**3071.28€**

BUDGET SUBTOTAL		**54,259.37€**
APPLICABLE VAT	21%	**11,394.47€**

TOTAL BUDGET	**65,653.84€**

C Code

This Annex lists the main parts of the code developed for this project.

C.1 Vector database deployment

Milvus deployment Docker Compose file.

```yaml
services:
    etcd:
        container_name: milvus-etcd
        image: quay.io/coreos/etcd:v3.5.5
        environment:
            - ETCD_AUTO_COMPACTION_MODE=revision
            - ETCD_AUTO_COMPACTION_RETENTION=1000
            - ETCD_QUOTA_BACKEND_BYTES=4294967296
            - ETCD_SNAPSHOT_COUNT=50000
        volumes:
            - ${MILVUS_DATA_PATH}/etcd:/etcd
        command: etcd -advertise-client-urls=http://127.0.0.1:2379 -
    listen-client-urls http://0.0.0.0:2379 --data-dir /etcd
        healthcheck:
            test: ["CMD", "etcdctl", "endpoint", "health"]
            interval: 30s
            timeout: 20s
            retries: 3

    minio:
        container_name: milvus-minio
        image: minio/minio:RELEASE.2023-03-20T20-16-18Z
        environment:
            MINIO_ACCESS_KEY: minioadmin
            MINIO_SECRET_KEY: minioadmin
        ports:
            - "9001:9001"
            - "9000:9000"
        volumes:
            - ${MILVUS_DATA_PATH}/minio:/minio_data
        command: minio server /minio_data --console-address ":9001"
        healthcheck:
            test: ["CMD", "curl", "-f", "http://localhost:9000/minio/
    health/live"]
            interval: 30s
            timeout: 20s
            retries: 3

    standalone:
        container_name: milvus-standalone
        image: milvusdb/milvus:v2.4.15
        command: ["milvus", "run", "standalone"]
        security_opt:
        - seccomp:unconfined
        environment:
            ETCD_ENDPOINTS: etcd:2379
            MINIO_ADDRESS: minio:9000
        volumes:
```

```
            - ${MILVUS_DATA_PATH}/milvus:/var/lib/milvus
        healthcheck:
            test: ["CMD", "curl", "-f", "http://localhost:9091/healthz"]
            interval: 30s
            start_period: 90s
            timeout: 20s
            retries: 3
        ports:
            - "19530:19530"
            - "9091:9091"
        depends_on:
            - "etcd"
            - "minio"
```

Qdrant deployment Docker Compose file.

```
services:
    qdrant:
        image: qdrant/qdrant
        ports:
            - 6333:6333
            - 6334:6334
        volumes:
            - ${QDRANT_DATA_PATH}:/qdrant/storage:z
```

C.2 Video Summarization Server

Milvus query route

```
// Add express
const express = require('express');
const http = require('http');

// Instantiate router object
const router = express.Router();

// Import milvus
const { MilvusClient, DataType } = require('@zilliz/milvus2-sdk-node');

// Add .env params
const dotenv = require('dotenv');
dotenv.config();

// To measure running times
const fs = require('fs');
const path = require('path');

function log_times(time, category) {
    const logFile = path.join('/logs', category+'_times.log');

    fs.appendFile(logFile, `${time}\n`, (err) => {
        if (err) console.log('Error with logging', err);
    })
}

// Route
```

```
router.get('/milvus', (req, res) => {

    const textQuery = req.query.text; // Get text to query
    const encoderQuery = req.query.encoder; // Get encoder to build the embedding
    const datasetQuery = req.query.dataset; // Get the dataset to query the database

    // If URL is wrong
    if(!textQuery || !encoderQuery || !datasetQuery) {
        return res.status(400);
    };

    let encoderName = encoderQuery;
    let collectionName;
    if (datasetQuery == 'centroid') {
        collectionName = 'uca'+encoderQuery+datasetQuery;
    } else {
        collectionName = 'ucf'+encoderQuery+datasetQuery;
    }
    console.log(collectionName)
    console.log(textQuery)

    // Get embedding
    embUrl = `http://${process.env.EMB_ENGINE_HOST}:${process.env.EMB_ENGINE_PORT}` +
        `/text?text=${textQuery}`;

    // POST request body
    const embData = JSON.stringify({
        encoder: encoderName,
        data: textQuery
    });

    // POST request options
    const embOptions = {
        hostname: process.env.EMB_ENGINE_HOST,
        port: process.env.EMB_ENGINE_PORT,
        path: '/text',
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Content-Length': embData.length
        }
    };

    const startEmb = Date.now();
    // Define callback for when the request is sent
    const embRequest = http.request(embOptions, embRes => {
        let embJSON = '';

        // Check for errors
        if (embRes.statusCode >= 400 && embRes.statusCode < 600) {
            // Capture the error message from the Flask server's response
            embRes.on('data', chunk => {
                embJSON += chunk.toString();
            });
```

```
82
83              embRes.on('end', () => {
84                  return res.status(500).json({
85                      status: embRes.statusCode.toString(),
86                      message: embJSON
87                  });
88              });
89
90              return;
91          }
92
93          // Callback for when data is being recieved
94          embRes.on('data', chunk => {
95              embJSON += chunk.toString(); // Store response as string as
    it comes by
96          });
97
98          // Callback for when data has been recieved
99          embRes.on('end', async () => {
100             try {
101                 const endEmb = Date.now();
102                 log_times(endEmb-startEmb, 'emb');
103
104                 let emb = JSON.parse(embJSON)
105                 console.log("Embedding recieved!")
106
107                 // Data to be sent in the POST request to Milvus
108                 let outputFields;
109                 if(collectionName.endsWith('centroid'))
110                     outputFields = ['video', 'start_frame', 'end_frame'
    ];
111                 else if(collectionName.endsWith('frames'))
112                     outputFields = ['video', 'frame_n']
113
114                 const startQuer = Date.now();
115                 const address = `http://${process.env.MILVUS_HOST}:${
    process.env.MILVUS_PORT}`;
116                 const token = "root:Milvus";
117
118                 console.log('Connecting to Milvus...')
119                 const client = new MilvusClient({address, token});
120
121                 console.log("Loading collection")
122                 await client.loadCollection({collection_name:
    collectionName})
123
124                 console.log("Querying Milvus...")
125                 let databaseData = await client.search({
126                     collection_name: collectionName,
127                     data: emb,
128                     limit: 10, // The number of results to return
129                     output_fields: outputFields
130                 });
131
132                 let resultsArray = []
133
134                 const endQuer = Date.now();
135                 log_times(endQuer-startQuer, 'query_milvus')
```

```
                console.log('Database queried')

                if(collectionName.endsWith('centroid')) {
                    databaseData.results.forEach(element => {
                        resultsArray.push({
                            video: element.video,
                            start_frame: element.start_frame,
                            end_frame: element.end_frame,
                        })
                    });
                } else if(collectionName.endsWith('frames')) {
                    databaseData.results.forEach(element => {
                        let start_frame = (element.frame_n < 75) ? 0 : (element.frame_n - 75);
                        let end_frame = element.frame_n + 75; // If not valid, streaming route fix it

                        resultsArray.push({
                            video: element.video,
                            start_frame: start_frame,
                            end_frame: end_frame,
                        })
                    });
                }

                console.log("Building results")
                const clientRes = {results: resultsArray}

                res.json(clientRes);


            } catch (err) {
                res.status(500);
            }
        });
    });

    embRequest.on("error", (err) => {
        console.log(err);
        res.status(500).json({status: "500", message: "Error with embedding server request"});
    })

    embRequest.write(embData);
    embRequest.end();
});

module.exports = router;
```

Qdrant query route

```
// Add express
const express = require('express');

// Instantiate router object
const router = express.Router();

// Add .env params
```

```
const dotenv = require('dotenv');
dotenv.config();

// Add HTTP library to build requests
const http = require('http');

//const got = require('got');

const fs = require('fs');
const path = require('path');

function log_times(time, category) {
    const logFile = path.join('/logs', category+'_times.log');

    fs.appendFile(logFile, `${time}\n`, (err) => {
        if (err) console.log('Error with logging', err);
    })
}

// To query QDrant Database
router.get('/qdrant', (req, res) => {

    const textQuery = req.query.text; // Get text to query
    const encoderQuery = req.query.encoder; // Get encoder to build the embedding
    const datasetQuery = req.query.dataset; // Get the dataset to query the database


    // If URL is wrong
    if(!textQuery || !encoderQuery || !datasetQuery) {
        return res.status(400);
    };

    let encoderName = encoderQuery;

    let collectionName;
    if (datasetQuery == 'centroid') {
        collectionName = 'uca'+encoderQuery+datasetQuery;
    } else {
        collectionName = 'ucf'+encoderQuery+datasetQuery;
    }
    console.log(collectionName)

    const startEmb = Date.now();
    // Get embedding
    embUrl = `http://${process.env.EMB_ENGINE_HOST}:${process.env.EMB_ENGINE_PORT}` +
        `/text`;

    // POST request body
    const embData = JSON.stringify({
        encoder: encoderName,
        data: textQuery
    });

    // POST request options
    const embOptions = {
```

```
        hostname: process.env.EMB_ENGINE_HOST,
        port: process.env.EMB_ENGINE_PORT,
        path: '/text',
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Content-Length': embData.length
        }
    };

    // Define callback for when the request is sent
    const embRequest = http.request(embOptions, embRes => {
        let embJSON = '';

        // Callback for when data is being recieved
        embRes.on('data', chunk => {
            embJSON += chunk.toString(); // Store response as string as it comes by
        });

        // Callback for when data has been recieved
        embRes.on('end', () => {
            const endEmb = Date.now();
            log_times(endEmb-startEmb, 'emb');
            try {
                // console.log("Sending embedding")
                let emb = embJSON

                // Data to be sent in the POST request body
                let outputFields;
                if(collectionName.endsWith('centroid'))
                    outputFields = ['video', 'start_frame', 'end_frame'];
                else if(collectionName.endsWith('frames'))
                    outputFields = ['video', 'frame_n']

                const postData = JSON.stringify({
                    query: JSON.parse(emb)[0],
                    limit: 10,
                    with_payload: outputFields
                });

                // Options for the POST request
                const options = {
                    hostname: process.env.QDRANT_HOST,
                    port: process.env.QDRANT_PORT,
                    path: `/collections/${collectionName}/points/query`,
                    method: 'POST',
                    headers: {
                    'Content-Type': 'application/json',
                    'Content-Length': Buffer.byteLength(postData)
                    }
                };

                const startQuer = Date.now();
                const databaseReq = http.request(options, databaseRes => {
                    let databaseData = '';
```

```
118
119                     // Collect data chunks as they arrive
120                     databaseRes.on('data', (chunk) => {
121                         databaseData += chunk;
122                     });
123
124                     // Process the complete response once all chunks are
    received
125                     databaseRes.on('end', () => {
126                         const endQuer = Date.now();
127                         log_times(endQuer-startQuer, 'query_qdrant');
128                         try {
129                             const results = JSON.parse(databaseData).
    result;
130
131                             let resultsArray = []
132
133                             if(collectionName.endsWith('centroid')) {
134                                 results.points.forEach(element => {
135                                     resultsArray.push({
136                                         video: element.payload.video,
137                                         start_frame: element.payload.
    start_frame,
138                                         end_frame: element.payload.
    end_frame,
139                                     })
140                                 });
141                             } else if(collectionName.endsWith('frames'))
     {
142                                 try {
143                                     results.points.forEach(element => {
144
145                                         let start_frame = (element.
    payload.frame_n < 75) ? 0 : (element.payload.frame_n - 75);
146                                         let end_frame = element.payload.
    frame_n + 75; // If not valid, streaming route fix it
147                                         // console.log(`Start: ${
    start_frame}`)
148                                         // console.log(`End: ${end_frame
    }`)
149                                         resultsArray.push({
150                                             video: element.payload.video
    ,
151                                             start_frame: start_frame,
152                                             end_frame: end_frame,
153                                         })
154                                     });
155                                 } catch (error) {
156                                     console.log(error)
157                                 }
158                             }
159                             console.log("Response built")
160                             const clientRes = {results: resultsArray}
161
162                             res.json(clientRes);
163                         } catch (error) {
164                             res.status(500)
165                         }
```

```
                    });
                });

                    // Handle errors in the QDrant request
                    databaseReq.on('error', (e) => {
                        res.status(500).json({ error: `QDrant request
    error: ${e.message}` });
                    });

                    // Write the data to the QDrant request body
                    databaseReq.write(postData);
                    databaseReq.end();
            } catch (err) {
                res.status(500);
            }
        });
    });

    embRequest.write(embData);
    embRequest.end();

});
```

Video trimming route

```
const express = require('express');

// File system
const fs = require('fs');

// Paths
const path = require('path');

// ffmpeg
const ffmpeg = require('fluent-ffmpeg');
const ffmpegStatic = require('ffmpeg-static');
const ffprobe = require('@ffprobe-installer/ffprobe');

// Set the path for ffmpeg
ffmpeg.setFfmpegPath(ffmpegStatic);
ffmpeg.setFfprobePath(ffprobe.path);  // Set ffprobe path

// Videos directory (inside the container)
const videoDirectory = '/videos'

function log_times(time, category) {
    const logFile = path.join('/logs', category+'_times.log');

    fs.appendFile(logFile, `${time}\n`, (err) => {
        if (err) console.log('Error with logging', err);
    })
}

const app = express.Router();

// Gets video frame rate
const getVideoFrameRate = (videoPath) => {
    return new Promise((resolve, reject) => {
```

```
        ffmpeg.ffprobe(videoPath, (err, metadata) => {
            if (err) {
                reject(err);
            } else {
                const frameRate = eval(metadata.streams[0].r_frame_rate); // Extract frame rate
                resolve(frameRate);
            }
        });
    });
};

// Gets video duration in frames
const getVideoLength = (videoPath) => {
    return new Promise((resolve, reject) => {
        ffmpeg.ffprobe(videoPath, (err, metadata) => {
            if (err) {
                reject(err);
            } else {
                const videoLength = eval(metadata.streams[0].duration_ts); // Extract video length
                resolve(videoLength);
            }
        });
    });
};

app.get('/', async (req, res) => {

    const { name, startFrame, endFrame } = req.query;

    const videoPath = path.join(videoDirectory, (name.endsWith('.mp4'))? name : name+'.mp4');
    console.log(videoPath)
    // Check if video exists
    if (!fs.existsSync(videoPath)) {
        return res.status(404).send('Video not found');
    }

    let startFrameNumber = parseInt(startFrame);
    let endFrameNumber = parseInt(endFrame);

    if (isNaN(startFrameNumber) || isNaN(endFrameNumber) || startFrameNumber >= endFrameNumber) {
        return res.status(400).send('Invalid startFrame or endFrame parameters');
    }

    const videoLength = await getVideoLength(videoPath);

    startFrameNumber = (startFrameNumber < 0) ? 0 : startFrameNumber;
    endFrameNumber = (endFrameNumber > videoLength) ? videoLength : endFrameNumber;

    try {
        // Get the video frame rate
        const frameRate = await getVideoFrameRate(videoPath);

```

```
        // Convert frame numbers to time (seconds)
        const startSeconds = startFrameNumber / frameRate;
        const endSeconds = endFrameNumber / frameRate;

        const startTrim = Date.now()
        // Use ffmpeg to trim and stream the video
        ffmpeg(videoPath)
            .setStartTime(startSeconds)
            .setDuration(endSeconds - startSeconds)
            .outputOptions('-movflags', 'frag_keyframe+empty_moov') // For better streaming support
            .toFormat('mp4')
            .on('error', (err) => {
                console.error('Error processing video:', err);
                res.status(500).send('Error processing video');
            })
            .on('end', () => {
                const endTrim = Date.now();
                log_times(endTrim-startTrim, 'trim')
            })
            .pipe(res, { end: true });

    } catch (err) {
        console.error('Error getting video frame rate:', err);
        res.status(500).send('Error processing video frame rate');
    }
});

module.exports = app;
```

C.3 Embedding Server

Main app script

```
from flask import Flask, request # type: ignore
import numpy as np # type: ignore

import base64
from PIL import Image
from io import BytesIO

possible_models = [
    'default',
    'random',
    'clip',
    'clip-centroid',
    'vclip'
]

from encoders import EncoderBuilder

app = Flask(__name__)

def check_request(req):
    return req['encoder'] and req['data']

```

```python
# This route takes a text and responds with the embedding
@app.route('/text', methods=['POST'])
def get_text():

    # Check if request includes all the necessary data
    if (not check_request(request.json)):
        return "Bad request", 400

    # Logging info
    print(f'Text encoding with: {request.json["encoder"]}')

    # Take the requested text
    text = request.json['data']

    try:
        # Build the selected model
        model = EncoderBuilder().build(request.json['encoder'])

        # Encode text
        emb = model.encode_text(text)
    except MemoryError:
        return "ERROR: CUDA out of memory", 500

    # Return succesful response
    return emb.tolist(), 200

# This route takes a b64-encoded image and responds with the embedding
@app.route('/image', methods=['POST'])
def get_image():

    if (not check_request(request.json)):
        return "Bad request", 400

    # Logging info
    print(f'Image encoding with: {request.json["encoder"]}')

    # Get and decode requested image
    img_b64 = request.json['data']
    image_data = base64.b64decode(img_b64)
    img = Image.open(BytesIO(image_data))
    img = np.array(img)

    try:
        # Build the selected model
        model = EncoderBuilder().build(request.json['encoder'])

        # Encode image
        emb = model.encode_image(img)
    except MemoryError:
        return "CUDA out of memory", 500

    # Return succesful response
    return emb.tolist(), 200

if __name__ == '__main__':
    app.run()
```

Encoders interface and handler

```python
from os.path import join as join_path
import torch

# For configuration parameters
from yacs.config import CfgNode as CN

# Usual CLIP
from clip import clip

# Modified CLIP
from vclip import vclip

possible_models = [
    'default',
    'random',
    'clip',
    'clip-centroid',
    'vclip'
]

class EmbeddingModel:
    def __init__(self):
        raise NotImplementedError

    def load(self):
        """Loads the model in cuda memory.
        """
        raise NotImplementedError

    def unload(self):
        """Unloads the model from cuda memory.
        """
        raise NotImplementedError

    def encode_image(self):
        """Generates the embedding of an image.
        """
        raise NotImplementedError

    def encode_text(self):
        """Generates the embedding of a text.
        """
        return NotImplementedError

    def encode_video(self):
        """Generates the embedding of a video.
        """
        return NotImplementedError

    def get_params(self) -> dict:
        """Returns the params related with the encoder, to properly
    configure a database's
        collection.

        Returns
        -------
        dict
            The parameters as a dictionary
```

```
        """
        raise NotImplementedError

class EncoderBuilder:

    def build(self, encoder_name, *args, **kwargs) -> EmbeddingModel:
        """Returns the encoder's object corresponding to the specified name.

        Parameters
        ----------
        encoder_name : str
            The name of the encoder. Can be one of the specified in the \'vision_encoders.possible_models\' variable.

        Returns
        -------
        EmbeddingModel
            The encoder's object.

        Raises
        ------
        TypeError
            If the specified encoder is not implemented.
        MemoryError
            If CUDA is out of memory.
        """
        try:
            match encoder_name:
                case 'clip':
                    return CLIP(*args, **kwargs)
                case 'vclip':
                    return VCLIP(*args, **kwargs)
                case _:
                    raise TypeError(f'TypeError: Encoder {encoder_name} not found among implemented. Please, use one of the following: {possible_models}.')
        except torch.OutOfMemoryError:
            raise MemoryError("CUDA out of memory")

class CLIP(EmbeddingModel):

    def __init__(self, model_name:str='openai/clip-vit-base-patch32'):
        """Uses CLIP model to obtain the embeddings of the clips.

        Parameters
        ----------
        model_name : str, optional
            The specific CLIP model.
        """
        self.model_name = model_name
        self.load()

    def load(self):
        try:
            self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
            self.model, self.processor = clip.load('ViT-B/32', device=
```

```
    self.device)
111             # self.model = CLIPModel.from_pretrained(self.model_name)
112             # self.model.to(self.device)
113             # self.processor = AutoProcessor.from_pretrained(self.
    model_name)
114         except OSError as e:
115             print(f'OSError: Model \'{self.model_name}\' not listed in
    HuggingFace repository. {e}')
116 
117     def encode_image(self, img):
118         inputs = self.processor(images=img, return_tensors='pt').to(self
    .device)
119 
120         image_features = self.model.get_image_features(**inputs)
121 
122         return image_features.cpu().detach().numpy()
123 
124     def encode_text(self, text):
125         tokenized_text = clip.tokenize(text).to(self.device)
126         # inputs = self.processor(text=text, return_tensors='pt').to(
    self.device)
127 
128         text_features = self.model.encode_text(tokenized_text)
129         # print(f'Text features: {text_features}')
130 
131         return text_features.cpu().detach().numpy()
132 
133     def get_encoder_params(self) -> dict:
134         params = {
135             'model_name': 'clip',
136             'embedding_size': 512,
137             'embedding_list': True
138         }
139         return params
140 
141 VCLIP_WEIGHTS_PATH = '/weights'
142 class VCLIP(EmbeddingModel):
143     def __init__(self):
144 
145         # Apply a default configuration
146         _C = CN()
147         _C.BASE = ['']
148         _C.MODEL = CN()
149         _C.MODEL.ARCH = 'ViT-B/32'
150         _C.MODEL.WEIGHTS_DIR = VCLIP_WEIGHTS_PATH
151         _C.MODEL.RESUME = join_path(VCLIP_WEIGHTS_PATH, '100
    batch_40frames_32.pth')
152         _C.DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
153         self.config = _C.clone()
154 
155         self.load()
156 
157     def load(self):
158         backbone_name = self.config.MODEL.ARCH
159         root = self.config.MODEL.WEIGHTS_DIR
160 
161         self.model, self.preprocess = vclip.load(name=backbone_name,
162                                                  download_root=root,
```

```
                                                    jit=False,
                                                    device=self.config.
    DEVICE)

        self.model = self.model.float().cuda()
        checkpoint = torch.load(self.config.MODEL.RESUME, map_location='
    cpu')
        load_state_dict = checkpoint['model']
        self.model.load_state_dict(load_state_dict, strict=False)

        self.device = self.config.DEVICE

    def unload(self):
        pass

    def encode_image(self, img):
        # Send image to device
        image = self.preprocess(img).unsqueeze(0).to(self.device)

        # Get features
        image_features, attention_weights = self.model.encode_image(
    image)

        return image_features

    def encode_text(self, text):
        text = vclip.tokenize(text).to(self.device)
        text_features, _ = self.model.encode_text(text)
        return text_features

    def get_encoder_params(self) -> dict:
        params = {
            'model_name': 'clip',
            'embedding_size': 768,
            'embedding_list': True
        }
        return params
```

C.4 Web code

HTML code

```
<!DOCTYPE html>
<html>
  <head>
    <title>GAPS - CCTV</title>
    <meta name="viewport" content="width=device-width, initial-scale=1"
    charset="UTF-8">
    <link rel="stylesheet" type="text/css" href="css/style.css">
    <link rel="icon" href="res/favicon.png" type="image/x-icon">
    <script type = "text/javascript" src="js/queryClient.js"></script>
  </head>
  <body>
    <div class="title"><h1>GAPS - CCTV</h1></div>

    <div id="searchDiv">
```

```
      <input id="searchBar" type="text" name="query" placeholder="
    Introduce search text..." required>
      <button id="searchButton" type="submit" onclick="querySearch();">
    Search</button>
      <script>
        document.getElementById('searchBar').addEventListener('keydown',
     function(event) {
          if (event.key === 'Enter') {
            querySearch();
          }
        });
      </script>
    </div>


    <div class="selection-container">
      <div class="selection">
          <label for="encoder">Choose an Encoder:</label>
          <select id="encoder" multiple>
              <option value="clip">CLIP</option>
          </select>
      </div>
      <div class="selection">
          <label for="dataset">Choose a dataset:</label>
          <select id="dataset" multiple>
              <option value="centroid">UCA</option>
              <option value="frames">UCF</option>
          </select>
      </div>
      <div class="selection">
          <label for="database">Choose a database:</label>
          <select id="database" multiple>
              <option value="milvus">Milvus</option>
              <option value="qdrant">QDrant</option>
          </select>
      </div>
  </div>
      <div class="results" id="results">
      </div>

    <div class="image-container">
      <img src="res/upmlogo.png" alt="Image 1">
      <img src="res/gapslogo.png" alt="Image 2">
    </div>
  </body>
</html>
```

CSS sstylesheet

```
body {
  font-family: "Lato", sans-serif;
  display: flex;
  flex-direction: column;
  align-items: center;
  height: 100vh;
  margin: 0;
}

```

```
#searchDiv {
  text-align:center;
  width:80%;
}

#searchBar {
  width: 70%;
  padding: 1%;
  font-size: 1em;
  border: 2px solid #ccc;
  border-radius: 5px;
}

#searchButton {
  background-color: rgb(80, 80, 165);
  color: white;
  transition: background-color 0.3s ease;
  padding: 1%;
  font-size: 1em;
  margin-left: 1%;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

#searchButton:hover {
  background-color: rgb(110, 110, 139);;
}

.title {
  margin-top: 2.5%;
  text-align:center;
}

.image-container {
  margin-top: 10%;
  text-align: center;
  background-color: #ccc;
  padding: 10px;
}

.image-container img {
  margin: 0 10px;
  width: 15%;
  height: auto;
}


.results {
  margin-top: 10%;
  display: flex;
  flex-direction: column;
  width: 80%;
}

.error-message {
  color: red;
  font-weight: bold;
```

```
  font-size: 1.5em;
  text-align: center;
  margin-top: 20px;
}


.entry {
  display: flex;
  justify-content: flex-start;
  align-items: center;
  margin-bottom: 5%;
  width: 100%;
  max-width: 800px;
}


.videoPlayer {
  width: 400px;
  height: auto;
  margin-right: 20px;
}


.video-info {
  font-size: 18px;
  line-height: 1.5;
  display: flex;
  flex-direction: column;
}

.video-info p {
  margin: 0;
  padding: 0;
}

.loader-container {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
}

.loader {
  border: 16px solid #f3f3f3;
  border-top: 16px solid #3498db;
  border-radius: 50%;
  width: 60px;
  height: 60px;
  animation: spin 2s linear infinite;
}

@keyframes spin {
  0% { transform: rotate(0deg); }
  100% { transform: rotate(360deg); }
}

.selection-container {
  display: flex;
```

```
  justify-content: space-around;
  margin-top: 1%;
  width: 30%;
  max-width: 100%;
  padding-left: 10%;
  padding-right: 10%;
}

.selection {
  text-align: center;
}

label {
  display: block;
  margin-bottom: 8px;
  font-size: 20px;
  font-weight: bold;
  line-height: 1.5;
}

select {
  padding: 8px;
  font-size: 16px;
}
```

Javascript code

```
function querySearch() {

    const resultsDiv = document.getElementById('results');


    // Get search text
    queryText = document.getElementById('searchBar').value;

    try {

        if(!queryText) {
            throw new Error('Please introduce a term in the search bar')
   ;
        }

        // Print loading wheel
        resultsDiv.innerHTML = '';
        const wheelCont = document.createElement('div');
        wheelCont.classList.add('loader-container')
        const wheel = document.createElement('div')
        wheel.classList.add('loader');
        wheelCont.appendChild(wheel)
        resultsDiv.appendChild(wheelCont)

        // Get encoder
        let encoderName = document.getElementById('encoder').value;
        let datasetName = document.getElementById('dataset').value;
        let databaseName = document.getElementById('database').value;

        if(!encoderName) {
            throw new Error('Please select an encoder');
```

```
        } else if(!datasetName) {
            throw new Error('Please select a dataset');
        } else if(!databaseName) {
            throw new Error('Please select a database');
        }

        // Url
        url = `http://localhost:3000/query/${databaseName}?text=${queryText}&encoder=${encoderName}&dataset=${datasetName}`

        fetch(url)
            .then(response => {
                if (!response.ok) {
                    response.json()
                        .then(err => {
                            let errorMessage = `Error sent from server: ${err.status} - ${err.message}`
                            resultsDiv.innerHTML = `<p class="error-message">${errorMessage}</p>`;
                        });
                    throw new Error("Error at querying");
                }
                return response.json();
            })
            .then(data => {
                printVideos(data);
            })
            .catch(err => {
                console.log(err);
            });
     } catch(err) {
        resultsDiv.innerHTML = `<p class="error-message">${err.message}</p>`;
     }

}

function printVideos(videosJson) {

    const resultsDiv = document.getElementById('results');
    resultsDiv.innerHTML = '';

    videosJson.results.forEach(element => {

        // Extract info
        let start_frame = element.start_frame;
        let end_frame = element.end_frame;
        let video_name = element.video;

        // Create entry division
        const entryDiv = document.createElement('div');
        entryDiv.classList.add('entry');

        // Create video element
        const videoPlayer = document.createElement('video');
        videoPlayer.classList.add('videoPlayer');
        videoPlayer.controls = true;
        videoPlayer.muted = true;
```

```
85
86          // Add source
87          const source = document.createElement('source');
88          source.src = `/video?name=${video_name}&startFrame=${start_frame
    }&endFrame=${end_frame}`;
89          source.type = 'video/mp4';
90          videoPlayer.appendChild(source);
91
92          // Create video metadata div
93          const videoInfoDiv = document.createElement('div');
94          videoInfoDiv.classList.add('video-info');
95
96          // Create camera name paragraph
97          const cameraInfo = document.createElement('p');
98          cameraInfo.textContent = `Video: ${video_name}`;
99
100         // Append paragraphs to videoInfoDiv
101         videoInfoDiv.appendChild(cameraInfo);
102
103         // Create the horizontal line (hr)
104         const hrElement = document.createElement('hr');
105
106         // Append video, info, and hr to the entry div
107         entryDiv.appendChild(videoPlayer);
108         entryDiv.appendChild(videoInfoDiv);
109         entryDiv.appendChild(hrElement);
110
111         // Append the entire entry div to the results div
112         resultsDiv.appendChild(entryDiv);
113
114     });
115 }
```