# StreamSense: Policy-driven Semantic Video Search in Streaming Systems

Gerard Finol, Arnau Gabriel, Pedro García-López
{name.surname}@urv.cat
Universitat Rovira i Virgili
Tarragona, Spain

Raúl Gracia-Tinedo
Luis Liu
{raul.gracia,luis_liu}@dell.com
Dell Technologies
Barcelona/Shanghai, Spain/China

Reuben Docea, Max Kirchner, Sebastian Bodenstedt
{name.surname}@nct-dresden.de
National Center for Tumor Diseases
Dresden, Germany

## ABSTRACT

Streaming systems are an increasingly appealing substrate for managing video data via the stream abstraction. However, if we consider a large stream collection, it can be hard for data scientists to discover and locate relevant videos, let alone specific video fragments. In this paper, we propose StreamSense: a policy-driven, semantic video search solution for streaming systems. StreamSense allows users to deploy AI models that generate embeddings from video frames via policies. Our system uses such embeddings for building a two-level index in a vector DB that efficiently handles inter/intra video queries. StreamSense abstracts users from vector DB interactions so they can perform semantic search using images as input and visualize the results. We built our prototype on top of a tiered streaming storage system (Pravega) and validated it on a health-related use case. We show that StreamSense allows data scientists to search for video fragments in real surgery datasets in $< 30$ms. StreamSense also reduces data ingestion related to AI training data loading in +80% compared to simple bulk loading video streams.

## CCS CONCEPTS

• **Information systems** → **Hierarchical storage management**; **Stream management**; **Video search**; • **Applied computing** → **Health informatics**.

## KEYWORDS

data streams, video analytics, semantic search, vector embeddings

**ACM Reference Format:**

Gerard Finol, Arnau Gabriel, Pedro García-López, Raúl Gracia-Tinedo, Luis Liu, and Reuben Docea, Max Kirchner, Sebastian Bodenstedt. 2024. StreamSense: Policy-driven Semantic Video Search in Streaming Systems. In *25th International Middleware Conference Industrial Track (MIDDLEWARE Industrial Track '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3700824.3701097

## 1 INTRODUCTION

Event streaming systems, such as Apache Kafka [4, 40], AWS Kinesis [1], or Apache Pulsar [5], are enabling organizations of all kinds to ingest, store, and process data in real-time with high performance and scalability. While streaming systems are often associated with managing event-like data types (*e.g.*, logs, sensors), there is an increasing interest in using these systems for managing multimedia. For instance, AWS Kinesis can ingest video streams and serve them in real-time to analytics applications [2]. Similarly, Pravega [13] offers a GStreamer connector for building video analytics pipelines [14]. Use cases like object identification in surveillance cameras, computer-assisted surgery, and quality control on manufacturing processes are just some examples that motivate streaming systems to support video streams as a first-class citizen [7, 11].

Furthermore, with the advent of AI, video streams are increasingly valuable not only when processed in streaming fashion, but also in batch. For instance, historical video data is a key asset in AI-related tasks like model training [47]. We also realize that this aligns naturally with the shift of streaming systems towards supporting storage tiering for data streams [6, 28]. Tiering stream data to a scale-out, cost-effective storage is an ideal solution when ingesting and storing video data for extended periods of time.

Unfortunately, while tiering stream data is a key requirement for managing historical video data, this alone is not enough. When training AI models, data scientists require the means to query and locate video streams, or even specific video fragments, within a large pool of videos. Such queries could be related to the content of videos, which goes beyond what video metadata can express. This requires some sort of semantic video search solution for streaming systems that, to our knowledge, is not available today.

### 1.1 Motivation and Challenges

Our main goal is to devise a flexible semantic video search solution for streaming systems. Achieving this goal could provide added value to data streaming platforms supporting video stream ingestion and analytics [1, 9]. For example, users could index video streams based on their own models and get accurate query results in the form of video fragments. Even more, data loaders in AI inference frameworks could exploit such a mechanism by ingesting only relevant video fragments for training a model, discarding the rest.

While promising, achieving this goal entails some challenges:

*(C1) Flexible content-based video stream indexing*: Although a video stream is immutable, its contents can be indexed in multiple ways. For instance, a data scientist may use a surgery video for training an AI inference model for liver segmentation, while another user may resort to the same video stream for learning surgery techniques with specific instruments. As one can infer, both users are interested in different fragments of the same video stream. But supporting queries for both users requires the system to understand


https://doi.org/10.1145/3700824.3701097

what a "liver" and a "surgical instrument" are in order to build an index accordingly. Therefore, our solution should enable users to (re)index the same video stream based on different models.

*(C2) Scalable semantic search*: Dealing with tiered video streams for long retention periods requires managing large amounts of index data. In use cases like health video analytics, we may require indexing each video stream with fine granularity (*e.g.*, embedding per key video frame). On the one hand, it seems apparent that building a global index for all the videos would not scale, as the amount of required memory per query is, in some cases, the size of the indexed vectors [45, 57]. On the other hand, the cost of querying individual per-stream indexes may grow linearly with the number of video streams. Thus, we need to devise an index management approach that provides a reasonable solution to this trade-off.

*(C3) Programmatic search interface*: We see potential in exposing a semantic search mechanism via APIs to external programs, in addition to serve data scientists. For instance, AI inference frameworks could reduce data transfers when loading data that is related to the specific model to train, instead of bulk loading a whole collection of video streams. However, this topic requires further exploration.

### 1.2 Contributions

In this paper, we present StreamSense: a policy-driven, semantic video search solution for streaming systems. StreamSense allows users to plug-in custom embedding models via policies for seamlessly indexing video streams both in real-time and batch. StreamSense builds a two-level indexing model: a global sampling index contains a small subset of key frame embeddings for all the videos, whereas the system also builds a full index per video stream/model pair associating vector embeddings with video frame offsets in a data stream. This allows us to efficiently run inter/intra video queries. StreamSense abstracts data scientists from vector DB interactions so they can perform semantic search using images as input and visualize the results. In summary, our contributions are:

- Design of StreamSense: a policy-driven semantic video search solution that exploits tiered storage in streaming systems.
- Implementation of StreamSense including index generation, index management, and semantic search interface.
- Evaluation of StreamSense on AWS with real workloads.

We built our prototype on top of a tiered streaming storage system (Pravega) and validated it on a computer-assisted surgery use case from the National Center for Tumor Diseases (NCT), Germany. Via experiments, StreamSense allows data scientists to search for video fragments in real video surgery datasets in $< 30$ms. StreamSense also reduces data ingestion related to AI training data loading (PyTorch) in +80% compared to simple bulk loading video streams.

## 2 BACKGROUND

In this section, we provide the essential background to understand the technologies and core design decisions of this work.

### 2.1 Pravega & GStreamer

Pravega [28] is a distributed, tiered storage system for data streams. Pravega stores *data events* in *streams*. A stream is a durable, elastic, append-only, unbounded sequence of bytes achieving good performance and consistency. Internally, streams are divided into *segments*. A stream segment is a partition of the data within a stream. A stream may have multiple segments open for appending events at a given time, which enables higher throughput [29].

Pravega offers client libraries implementing the server APIs, such as *writers* and *readers*. On the server side, we find the Pravega *control plane* formed by *controller instances*. The control plane is responsible for orchestrating stream lifecycle operations, as well as managing the system's metadata. The *data plane* handles IO requests from clients and is formed by *segment store instances*. The segment store has two storage tiers: *Write-Ahead Log (WAL)* and *Long-Term Storage (LTS)*. The main goal of WAL (implemented via Apache Bookkeeper [3, 36]) is to guarantee durability and low latency of incoming writes and keep that data temporarily for recovery purposes. Segment stores asynchronously move data to LTS, which acts as the final destination of stream data.

For supporting video analytics, Pravega provides a GStreamer connector [14]. With GStreamer [10], users can build multimedia pipelines and streaming applications. Our GStreamer connector exposes the ability to read and write byte buffers corresponding to video frames in GStreamer pipelines. Thus, Pravega is a durable `sink` and a low-latency `source` for GStreamer applications.

### 2.2 Computer-assisted Surgery in NCT

The National Center for Tumor Diseases (NCT, Germany) [12] is an institution that mixes data scientists and surgeons to apply AI techniques on surgery-related multimedia. Specifically, we are collaborating with NCT to supporting a computer-assisted surgery use case. NCT requires video data from surgery cameras to be durably ingested and processed in real time via specialized AI inference models to help surgeons during the procedure [42]. Moreover, video data should be durably stored in long-term storage, so it can be accessed via batch analytics (*e.g.*, AI model training). We have built a proof-of-concept for NCT by using Pravega and GStreamer for managing video streams and feeding containerized AI jobs.

Nonetheless, a new challenging requirement for NCT is to search for specific video fragments, or even individual frames, in a collection of video streams. This is required for multiple reasons, ranging from the creation of specialized AI model training datasets to help surgeons or medical students locating specific video fragments of certain anatomies. If we consider a large collection of video streams, using just the metadata of video streams is not enough to satisfy content-based queries. Moreover, data scientists may require to find videos based on unstructured data as input, like an image. We need a content-based approach for indexing, querying, and retrieving relevant video stream data in streaming systems.

## 3 STREAMSENSE DESIGN

In what follows, we describe the design of StreamSense: our policy-driven semantic video search solution for streaming systems.

### 3.1 Policy-driven Video Embeddings Generation

In Fig. 1, we provide a high-level overview of the architecture of StreamSense and the use case that motivates it. First, we identify the video ingestion phase. As visible in step ① of Fig. 1, video frames from surgery cameras are ingested in Pravega as *video streams*. Pravega achieves low latency and durability upon video ingestion

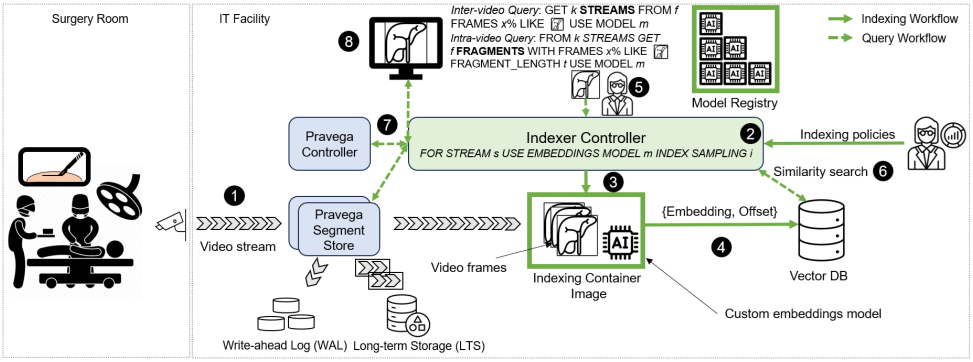



**Figure 1: Overview of StreamSense design and main components in the NCT use case.**

thanks to the WAL [36], which is expected for streaming computations. Moreover, the storage tiering mechanism in Pravega automatically moves video data to LTS. Upon a batch job processing video data, Pravega will exploit the high read throughput and parallelism of LTS systems (*e.g.*, NFS, AWS S3, HDFS). Therefore, using a tiered streaming system like Pravega achieves a sweet spot in the latency vs throughput trade-off when ingesting video data.

The *indexer controller* is the main component of StreamSense. One of its tasks is to orchestrate indexing activities via *policies* [30]. To this end, users can define policies for indexing video streams (step ② in Fig. 1). Such policies are expressive enough for capturing relevant aspects of the indexing process, like the embedding model to be used or the index sampling granularity. For example:

```
FOR STREAM s USE EMBEDDINGS MODEL m INDEX SAMPLING i
```

With the policy above, the data scientist instructs the system to index a video stream $s$ via the embeddings model $m$, as well as to use an index sampling algorithm $i$ (see §3.2). The indexing controller keeps the information of policies in a metadata store.

Indexing policies may be in two states: *active* and *completed*, depending on whether the indexing process for a video stream has finished or not. The indexer controller triggers the embedding generation process for active policies. Retaking the previous example, the indexer controller spawns a new *indexing instance* (*e.g.*, container or VM) with the requested embedding model running inside and consuming data from stream $s$ (step ③ in Fig. 1). Note that policies apply to both real time and historical video streams. Exploiting tiered data streams abstracts stream readers from the data location, while achieving good performance in both streaming and batch computations. With StreamSense, a data scientist could trigger multiple indexing instances for the same video stream, thus generating multiple semantic indexes for the same content.

For simplicity, StreamSense provides an *indexing container image* that handles IO with the streaming system and contains the necessary dependencies. StreamSense creates a new indexing instance using the indexing container image with the appropriate model from the *model registry* (*i.e.*, repository of embeddings models on top the indexing container image). Under the hood, the indexing container image ingests the video frames, passes them through the model, and writes the index data in the vector DB.

## 3.2 Two-level Video Indexing

In health use-cases like NCT, every frame may contain critical information pertaining to surgical events, such as serious complications. Missing video frames could result in overlooking vital details, which motivates us to store full video stream indexes (step ④ in Fig. 1). However, while vector DBs can create indexes over large vector embeddings collections, performing efficient queries on a single index containing all the video embeddings can be challenging [25]. For example, vector DBs like Milvus [55] are limited in their query capabilities to the amount of available memory [45]. On the other hand, just creating an index per video stream and embeddings model can be problematic too. To wit, since each query can only access one video stream index, the cost of searching for content may grow in the order of the number of video stream indexes. This is an interesting trade-off that calls for a vector DB-agnostic solution.

In StreamSense, we build a two-level video stream indexing layout per embeddings model (see Fig. 2). First, indexing policies define an *indexing sampling algorithm*. StreamSense currently offers time-based index sampling (*e.g.*, index a frame every 1 or 30 minutes), but more sophisticated "shot boundary" algorithms can be added in the future [52, 60]. New video sampling algorithms just need to implement the `do_sampling(frame, embedding, offset)` method in the indexing container image. The index sampling algorithm determines the embeddings that will be stored in the *stream sampling index* (via the `add_to_sampling_index(embedding, stream)` call). Such an index will contain a small set of embeddings for all the videos in the system associated to the video stream. The stream sampling index enables StreamSense to search for videos in which *any* (sampled) frame satisfies a similarity query.

An indexing instance also builds a *full index* for the stream at hand (*e.g.*, one embedding per key frame). The outcome of the embeddings model running in the indexing instance is written to the vector DB via the `add_to_index(stream, embedding, frame_id, offset)` call. With this index, StreamSense can perform fine-grained search on any video stream. This is key for finding relevant video fragments given a specific similarity criterion.

In summary, our two-level video indexing layout trades-off a small storage penalty building the sampling index (*e.g.*, 1% to 5% additional vector embeddings depending on the sampling model) for better query scalability (see results in §4.3). Next, we describe how StreamSense uses this index for serving semantic queries.

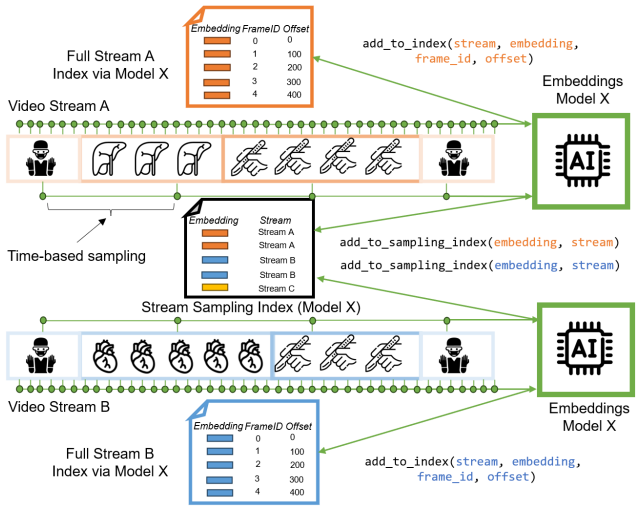



**Figure 2: The stream sampling index helps us to find video streams in which any of their computed sample embeddings matches a similarity query. A per-stream index allows us running semantic search within video streams.**

### 3.3 Semantic Video Search

The indexer controller also takes care of orchestrating the query workflow in StreamSense. First, it receives the input data for the query at hand. We consider two types of input data: *vector embeddings* and *images*. In the former, the provided embedding is directly used to perform the similarity search against the vector DB. For the latter, the indexer controller should first generate the embedding from the input image and then perform the similarity search.

Based on the generated indexes, StreamSense allows users to perform two types of queries: *inter-video* and *intra-video* (step ⑤ in Fig. 1). Queries target the index generated from the embeddings model $m$ (step ⑥ in Fig. 1). Let's see an inter-video query example:

```
GET k STREAMS FROM f FRAMES x% LIKE [IMAGE] USE EMBEDDINGS MODEL m
```

Inter-video queries use the sampling embeddings index for the given embeddings model and enable users finding video streams in which any (sampled) frame matches the similarity criterion. Inter-video queries expose four parameters to users: i) the maximum number of video streams to return to the user ($k$), ii) the number of closest sample frames to retrieve from the vector DB ($f$), iii) the similarity threshold or percentage required for result video streams ($x$), and iv) the embeddings model ($m$). The query workflow boils down to retrieving from the sampling embeddings index for model $m$ the top $k \cdot f$ sample frame embeddings most similar to the input embedding. We filter out the embeddings that do not meet the similarity threshold $x$ for retrieving the relevant best $k$ streams.

Intra-video queries allow users finding fragments within a video stream that match the similarity criterion. Let's see a query example:

```
FROM k STREAMS GET f FRAGMENTS WITH FRAMES x%
LIKE [IMAGE] FRAGMENT_LENGTH t USE EMBEDDINGS MODEL m
```

The query workflow works in two steps: first, the indexer controller runs an inter-video query with the input embedding to retrieve the most similar $k$ video stream candidates and discarding the ones not respecting the similarity threshold $x$. Then, with the resulting video stream collection, the indexer controller will search the specific video stream full indexes for the $f$ most similar frames —described by `<embedding:frame_id:offset>` triples— to the input image reference. For each video stream, the query filters the `<embedding:frame_id:offset>` triples with an accuracy that meets the embedding similarity threshold $x$. The length of the video fragment surrounding a relevant frame is determined by the query parameter $t$ (*e.g.*, 20 seconds). The cost in terms of vector DB queries of this process is 1 query to the sampling embeddings index and up to $k$ queries to find frames to the individual video stream full indexes, which can be parallelized for better performance. Note that it may be the case for several relevant frames to be consecutive in a video stream. To minimize redundant results, video fragments with overlapping offsets are merged into a single one.

Both inter/intra-video queries are exposed via APIs in the indexer controller (*i.e.*, `find_streams(image, k, f, x, m)` and `find_fragments(image, k, f, x, t, m)`). This enables external programs to exploit the semantic video search in StreamSense. We believe the use-case of collecting relevant datasets for AI training in inference frameworks (*e.g.*, PyTorch) is especially interesting. As we assess in §4.4, this allows users to easily find and load relevant training datasets, while reducing data transfers during the process.

Finally, the indexer controller gets the video stream names or video fragments offsets from a query and interacts with Pravega for displaying the query results back to the data scientist (step ⑦ in Fig. 1). StreamSense also provides facilities for visualizing the outcomes of queries, thus providing a full end-to-end solution for semantic video search (step ⑧ in Fig. 1).

## 4 EVALUATION

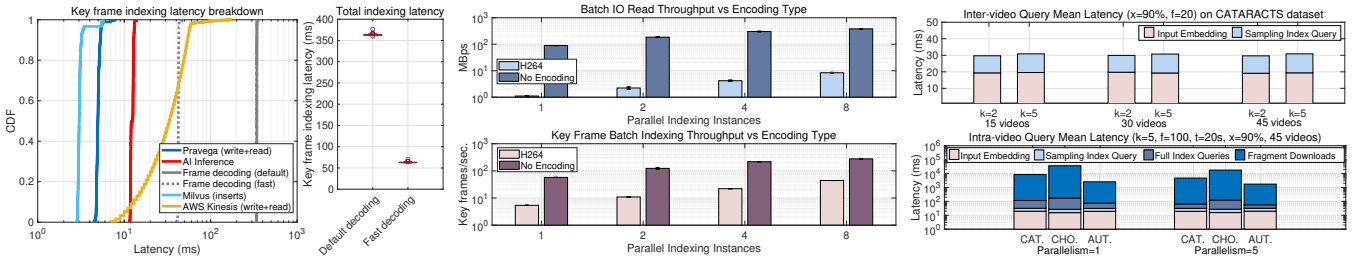
Our evaluation focuses on: i) what is the video indexing performance of StreamSense? (§4.2), ii) what is the semantic search latency of video streams/fragments? (§4.3), iii) can StreamSense reduce data loading phase for AI training via semantic search? (§4.4).

### 4.1 Setup

We summarize here the configuration used in our AWS experiments.

*Prototype Implementation.* Our implementation relies on Pravega for video stream IO via the GStreamer connector [14] and Milvus [31, 55] as a vector DB. We also have a base docker image with the necessary dependencies for managing video streams from Pravega which are consumed by AI inference models. The indexer controller is implemented in Python and manages indexing policies and the life-cycle of indexing instances (*e.g*, as VMs or Kubernetes pods). The code StreamSense is publicly available [18].

*Deployment.* StreamSense is deployed in a EC2 cluster of 4 nodes (`us-east-1a`) inside the same VPC. Pravega uses an `i3en.2xlarge` instance (8 vCPUs, 64GB of RAM, and 2 NVMe) running a Pravega controller, a Pravega segment store, a Bookkeeper instance, and a Zookeeper instance. We use an EFS share as long-term storage for Pravega. The indexer controller runs on a `p3.2xlarge` instance (8 vCPUs, 61GB of RAM, and an NVIDIA V100 GPU) jointly with




(a) Key frame indexing latency. (b) Batch key frame indexing throughput. (c) Semantic query latency.

**Figure 3: StreamSense performance: i) key frame streaming indexing latency with `h264` encoding (left), ii) throughput indexing parallel streams in batch depending on the encoding type (center), and iii) inter/intra stream semantic query latency (right).**

the AI inference model. An official standalone Milvus hosted on a `m5.2xlarge` instance (8 vCPUs, 32GB of RAM) configured with a global `Collection` that will store the embeddings from the sampled frames, and a `Collection` per video stream that will store the embeddings from all the key frames (*e.g.*, 1 per second). In Milvus, we use the `Inverted File Index` index type with 64-point clusters and the cosine similarity metric for the search and bounded `staleness` consistency mode (default). A `c5.4xlarge` VM (16 vCPUs, 32GB of RAM) generates `h264` (25FPS, 940x560) video streams emulating surgery cameras. We run a GStreamer pipeline that reads MP4 files from our datasets and writes them to Pravega as streams.

*Datasets and Embedding Models.* We utilize three publicly available datasets of surgical videos: CATARACTS [21], which includes videos of cataract surgeries; CHOLEC80 [46], which comprises videos of cholecystectomy surgeries; and AUTOLAPARO [56], which consists of videos of laparoscopic hysterectomy surgeries. To extract the embeddings, we make use of ResNet50 [32] trained with the `IMAGENET1K_V1` dataset for its performance in our similarity use case. It is also widely used for surgery processes, as seen in [37, 59].

## 4.2 Video Indexing Performance

Next, we focus on the indexing performance of StreamSense both in streaming and batch (Fig. 3). In the streaming case, Fig. 3(a) the total indexing latency for key video frames and the latency breakdown. Indexing latency is measured by time-stamping video frames at the source and calculating the delta through the indexing process.

In Fig. 3(a), the total key frame indexing latency ranges between 63ms and 360ms in average, depending on the decoding configuration. This is the waiting time for a data scientist to perform semantic search on ingested video frames in StreamSense. We observe that video decoding is a compute-intensive task that seems to dominate indexing latency. A plausible explanation to this is that, while inference models use the available VM GPU, GStreamer decoding by default uses CPU. Some decoding configurations also entail buffering frames, which has a toll on latency as well. The latency difference between the default GStreamer `decodebin` and the `fast` configuration (*i.e.*, `ultrafast` speed preset, `fastdecode`) gives a sense on the room for optimization in this indexing stage.

Fig. 3(a) shows a latency breakdown indexing key frames in StreamSense. The Pravega IO latency shows a p99 latency under 7ms. This confirms that Pravega achieves good performance for managing video data in real-time, while providing automated storage tiering to LTS. For comparison, we have performed a micro-benchmark measuring the latency of writing and reading 10KB events in AWS Kinesis at a rate of 25 events/second (same rate as our video streams). Visibly, Kinesis exhibits a higher IO latency than Pravega (*e.g.*, 10.7x higher latency at p95) and does not provide storage tiering. The AI inference latency to generate embeddings takes around 12-14ms per key frame and inserts to Milvus take 3-6ms. This latency meets the requirements for streaming workloads. Again, the video decoding phase dominates latency (30-330ms).

Next, we focus on the batch indexing throughput depending on the video encoding configuration. Fig. 3(b) compares the indexing throughput of parallel VMs depending on if video uses `h264` encoding (`fast` configuration) or no encoding (*i.e.*, raw frames). Visibly, there is an interesting trade-off between indexing throughput and data storage/network efficiency. With `h264` encoding, VMs index video at a rate of 8.5MBps after decoding (5.5 key frames/second), which translates to $\approx$ 1MBps of read throughput ($8x$ transfer reduction). On the other hand, an indexing VM with the `no_encoding` option achieves an indexing throughput of $\approx$ 55 key frames/second thanks to avoiding the decoding overhead, which translates into 85MBps at the IO level. In the 8-VM case, the Pravega instance is saturated at 400MBps. However, the raw video storage size is around 200x larger compared to `h264` encoded video. StreamSense lets users to pick the best encoding based on their needs.

## 4.3 Semantic Video Search Latency

Next, we evaluate the latency of inter/intra video queries in StreamSense. The upper plot in Fig. 3(c) shows the latency of querying the stream sampling index to find streams ($k = 2$, $k = 5$) whose sampled frames meet a certain similarity threshold ($x = 90\%$) for various dataset sizes. Visibly, neither the tested number of streams to retrieve ($k$) nor the tested dataset sizes (15GB to 50GB) seem to have an impact on mean query latency, which is around 30ms. Most of the inter-video query time is spent on generating the input image embedding ($\approx$ 19ms), whereas $3 - 4$ms are related to Milvus query and the rest to filtering query results. This is mainly because the sampling index grows slowly, as the sampling interval is set to 30 seconds. We have micro-benchmarked (VectorDBBench [19]) our Milvus instance and found that the sampling index exhibits similarity search latency > 15ms with 0.5 million embeddings. With

| 15 *videos* | CATARCT query | CHOLEC query | LAPARO query |
|---|---|---|---|
| $k = 2, f = 50$ | 89.12% | 97.13% | 99.77% |
| $k = 5, f = 100$ | 83.79% | 87.97% | 99.77% |

| 45 *videos* | CATARCT query | CHOLEC query | LAPARO query |
|---|---|---|---|
| $k = 2, f = 50$ | 97.00% | 98.38% | 99.83% |
| $k = 5, f = 100$ | 93.94% | 92.82% | 99.79% |

**Table 1: Data transfer reduction of downloading relevant intra-video query fragments vs transferring full dataset.**

the same sampling interval, that would allow us storing 4.2k video hours before sampling index queries take two digit milliseconds.

The lower plot in Fig. 3(c) shows the intra-query latency per dataset downloading $f = 100$ video fragments of $t = 20$ seconds from the top $k = 5$ video streams that meet a $x = 90\%$ similarity. As expected, depending on the dataset, > 98% of query time is related to downloading video fragments from Pravega. This is reasonable as the amount of data to be downloaded ranges from 6MBs to 160MBs. Still, we observe that increasing query parallelism from 1 to 5 threads reduces intra-video query execution in 32% to 51%.

### 4.4 Data Transfer Savings in AI Data Loading

In this experiment, we address the problem of data scientists downloading relevant video data for training their AI models. In Table 1, we exercise the preliminary integration our PyTorch data loader for StreamSense. We allow the data loader to use an embedding as input —as well as the other intra-video query parameters— and calling the StreamSense service for downloading similar video fragments. As can be observed, StreamSense allows data scientists to automatically load relevant data for training their AI models while achieving a transfer reduction from 83.79% up to 99.83% compared to bulk transferring the whole dataset, depending on the intra-video query parameters ($k$ and $f$) and the dataset.

## 5 RELATED WORK

Event streaming systems are the foundation for a wide range of big data infrastructures [8, 9, 17, 20]. Systems like Kafka [4, 24], Pulsar [5], Kinesis [1], and more recently, Red Panda [15] are increasingly popular for managing stream data and serve processing engines in a variety of scenarios [23]. In this work, we are especially interested in exploiting the streaming systems for managing video data. To this end, we identify two key features that are desirable: storage tiering and video libraries. On the former, Pulsar [6] and Red Panda [16] have implemented storage tiering mechanisms for stream data. Povzner *et al.* [49] proposed Kora, a cloud-native streaming platform based on Kafka that provides storage tiering. On the latter, Amazon Kinesis [1] is one popular example of a streaming system with advanced libraries for video storage and processing [2]. StreamSense builds upon Pravega, which pioneered the shift from event streaming to streaming storage [13, 28]. Combined with the GStreamer connector [10, 14], Pravega is a powerful streaming storage substrate for building video ingestion and processing pipelines.

While there are multiple works focused on system challenges related to video analytics [26, 27, 41, 48, 53, 61], exploiting streaming systems for managing and processing video data remains largely unexplored [50]. Authors in [38] create a prototype of an streaming image recognition framework on top of Ray and Apache Kafka. In [54], authors propose a video analytics framework for Edge environments that builds upon the concept of virtual function chains for running computations on video data (*i.e.,* using Kafka and Spark). Saoudi *et al.* [51] exploit machine learning techniques to extract key frames for rapid browsing and efficient video indexing in Apache Storm. Perhaps, the closest work to the present one is that of Ichinose *et al.* [34]. In this work, authors propose an evaluation framework based on ingesting videos from multiple cameras via Kafka and analyzing them using Spark. Compared to prior art, StreamSense exploits tiered data streams for providing seamless semantic video indexing both in real-time and batch.

A core goal of this work is to allow content-based video search on data streams [33]. In this regard, Chang *et al.* [22] designed one of the first content-based video search engines —namely, VideoQ— supporting automatic object-based indexing and spatio-temporal queries. Jian *et al.* [35] aim at scaling content-based video search by identifying consistent concepts that can be efficiently indexed via a modified inverted index. More recently, authors in [44] propose a new method for transforming concept-based key frame and query representations into a common semantic embedding space, which aligns with the spirit of this work. Authors in [43] propose DeepStore: an in-storage accelerator architecture for supporting AI/ML content-based search. Yoon *et al.* [58] propose a content-based video retrieval method based on prototypical category approximation. In a similar context to ours, Surch [39] is a content-based semantic video search system for surgical scenarios. Compared to these works, StreamSense does not advocate for a specific algorithm capturing video features. Instead, we provide a policy-based solution that allows users (re)processing video streams based on specialized embeddings models on top of a streaming storage infrastructure.

## 6 CONCLUSIONS AND FUTURE WORK

This work presents StreamSense: a policy-driven semantic video search system for streaming systems. A key insight in the design of StreamSense is to exploit tiered data streams for management video data both in real-time and batch. StreamSense allows users to deploy custom embeddings models via policies and generate multiple indexes per video stream for content-based search. Our system also builds a two-level indexing layout for scaling inter/intra video queries. In our experiments on AWS, we observed that StreamSense exhibits high indexing performance with minimal intervention of the data scientist. Content-based video queries for surgery datasets can be executed in < 30ms. Furthermore, StreamSense can be used for reducing data transfers in +80% from AI training models during data loading (PyTorch) by importing just relevant videos for training a specific model at hand. Our future plans include exploiting specialized libraries for GPU-powered video encoding, exploring the accuracy of dynamic key frame sampling algorithms, and applying our system to other use cases (*e.g.*, manufacturing, surveillance).

## ACKNOWLEDGMENTS

We thank the Cloud Native Computing Foundation (CNCF) for sponsoring the Pravega project and the Pravega open-source community. This work has received funding from the European Union Horizon Europe research and innovation programme under grant agreements 101092644 (NEARDATA) and 101092646 (CLOUDSKIN).

# REFERENCES


[1] 2024. Amazon Kinesis. https://aws.amazon.com/es/kinesis.
[2] 2024. Amazon Kinesis Video Streams examples. https://docs.aws.amazon.com/kinesisvideostreams/latest/dg/examples.html.
[3] 2024. Apache Bookkeeper. https://bookkeeper.apache.org.
[4] 2024. Apache Kafka. https://kafka.apache.org.
[5] 2024. Apache Pulsar. https://pulsar.apache.org.
[6] 2024. Apache Pulsar - Overview of tiered storage. https://pulsar.apache.org/docs/2.11.x/tiered-storage-overview.
[7] 2024. Azure AI Video Indexer. https://azure.microsoft.com/en-us/products/ai-video-indexer.
[8] 2024. Confluent. https://www.confluent.io/.
[9] 2024. Dell Streaming Data Platform. https://www.dell.com/en-us/dt/storage/streaming-data-platform.htm.
[10] 2024. GStreamer. https://gstreamer.freedesktop.org/.
[11] 2024. Guidance for Semantic Video Search on AWS. https://aws.amazon.com/es/solutions/guidance/semantic-video-search-on-aws/.
[12] 2024. National Centre for Tumor Diseases (NCT) in Heidelberg. https://www.nct-heidelberg.de/en/the-nct.html.
[13] 2024. Pravega. https://cncf.pravega.io.
[14] 2024. Pravega - GStreamer Connector. https://github.com/pravega/gstreamer-pravega.
[15] 2024. RedPanda. https://redpanda.com.
[16] 2024. RedPanda - Tiered Storage. https://docs.redpanda.com/current/manage/tiered-storage/.
[17] 2024. StreamNative. https://streamnative.io/.
[18] 2024. StreamSense code repository. https://github.com/neardata-eu/video-stream-indexing.
[19] 2024. VectorDBBench: A Benchmark Tool for VectorDB. https://github.com/zilliztech/VectorDBBench.
[20] 2024. Ververica. https://www.ververica.com/.
[21] Hassan Alhajj, Mathieu Lamard, Pierre-henri Conze, Béatrice Cochener, and Gwenolé Quellec. 2021. CATARACTS. https://doi.org/10.21227/ac97-8m18
[22] Shih-Fu Chang, William Chen, Horace J Meng, Hari Sundaram, and Di Zhong. 1998. A fully automated content-based video search engine supporting spatiotemporal queries. *IEEE Transactions on Circuits and Systems for Video Technology* 8, 5 (1998), 602–615.
[23] Miyuru Dayarathna and Srinath Perera. 2018. Recent advancements in event processing. *Comput. Surveys* 51, 2 (2018), 1–36.
[24] Philippe Dobbelaere and Kyumars Sheykh Esmaili. 2017. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In *ACM DEBS'17*. 227–238.
[25] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2020. Scalable machine learning on high-dimensional vectors: From data series to deep network embeddings. In *WIMS'20*. 1–6.
[26] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *USENIX NSDI'17*. 363–376.
[27] Ila Gokarn, Hemanth Sabbella, Yigong Hu, Tarek Abdelzaher, and Archan Misra. 2023. MOSAIC: Spatially-multiplexed edge AI optimization over multiple concurrent video sensing streams. In *ACM MMSys'23*. 278–288.
[28] Raúl Gracia-Tinedo, Flavio Junqueira, Tom Kaitchuck, and Sachin Joshi. 2023. Pravega: A Tiered Storage System for Data Streams. In *ACM/IFIP Middleware'23*. 165–177.
[29] Raúl Gracia-Tinedo, Flavio Junqueira, Brian Zhou, Yimin Xiong, and Luis Liu. 2023. Practical Storage-Compute Elasticity for Stream Data Processing. In *ACM/IFIP Middleware'23 (Industrial Track)*. 1–7.
[30] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. 2017. Crystal: Software-Defined Storage for Multi-Tenant Object Stores. In *USENIX FAST'17*. 243–256.
[31] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. 2022. Manu: a cloud native vector database management system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3548–3561.
[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]
[33] Weiming Hu, Nianhua Xie, Li Li, Xianglin Zeng, and Stephen Maybank. 2011. A survey on visual content-based video indexing and retrieval. *IEEE Transactions on Systems, Man, and Cybernetics* 41, 6 (2011), 797–819.
[34] Ayae Ichinose, Atsuko Takefusa, Hidemoto Nakada, and Masato Oguchi. 2017. A study of a video analysis framework using Kafka and spark streaming. In *IEEE Big Data'17*. IEEE, 2396–2401.
[35] Lu Jiang, Shoou-I Yu, Deyu Meng, Yi Yang, Teruko Mitamura, and Alexander G Hauptmann. 2015. Fast and accurate content-based semantic search in 100m internet videos. In *ACM MM'15*. 49–58.
[36] Flavio P Junqueira, Ivan Kelly, and Benjamin Reed. 2013. Durability with bookkeeper. *ACM SIGOPS operating systems review* 47, 1 (2013), 9–15.
[37] Lukasz Kaiser, Aidan N. Gomez, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones, and Jakob Uszkoreit. 2017. One Model To Learn Them All. arXiv:1706.05137 [cs.LG]
[38] Kasumi Kato, Atsuko Takefusa, Hidemoto Nakada, and Masato Oguchi. 2019. Construction Scheme of a Scalable Distributed Stream Processing Infrastructure Using Ray and Apache Kafka. In *CATA'19*. 368–377.
[39] Jeongyeon Kim, Daeun Choi, Nicole Lee, Matt Beane, and Juho Kim. 2023. Surch: Enabling structural search and comparison for surgical videos. In *ACM CHI'23*. 1–17.
[40] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *NetDB'11*, Vol. 11. 1–7.
[41] Shengzhong Liu, Tianshi Wang, Hongpeng Guo, Xinzhe Fu, Philip David, Maggie Wigness, Archan Misra, and Tarek Abdelzaher. 2022. Multi-view scheduling of onboard live video analytics to minimize frame processing latency. In *IEEE ICDCS'22*. IEEE, 503–514.
[42] Lena Maier-Hein et al. 2017. Surgical data science for next-generation interventions. *Nature Biomedical Engineering* 1, 9 (2017), 691–696.
[43] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wenmei Hwu. 2019. Deepstore: In-storage acceleration for intelligent queries. In *IEEE/ACM MICRO'19*. 224–238.
[44] Foteini Markatopoulou, Damianos Galanopoulos, Vasileios Mezaris, and Ioannis Patras. 2017. Query and keyframe representations for ad-hoc video search. In *ACM ICMR'17*. 407–411.
[45] Milvus. 2024. Product FAQ. https://milvus.io/docs/product_faq.md#What-is-the-maximum-dataset-size-Milvus-can-handle.
[46] Chinedu Innocent Nwoye et al. 2022. Rendezvous: Attention mechanisms for the recognition of surgical action triplets in endoscopic videos. *Medical Image Analysis* 78 (2022), 102433. https://doi.org/10.1016/j.media.2022.102433
[47] David Ouyang, Bryan He, Amirata Ghorbani, Neal Yuan, Joseph Ebinger, Curtis P Langlotz, Paul A Heidenreich, Robert A Harrington, David H Liang, Euan A Ashley, et al. 2020. Video-based AI for beat-to-beat assessment of cardiac function. *Nature* 580, 7802 (2020), 252–256.
[48] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. 2023. Gemel: Model Merging for {Memory-Efficient},{Real-Time} Video Analytics at the Edge. In *USENIX NSDI'23*. 973–994.
[49] Anna Povzner, Prince Mahajan, Jason Gustafson, Jun Rao, Ismael Juma, Feng Min, Shriram Sridharan, Nikhil Bhatia, Gopi Attaluri, Adithya Chandra, et al. 2023. Kora: A Cloud-Native Event Streaming Platform for Kafka. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3822–3834.
[50] Theofanis P Raptis and Andrea Passarella. 2023. A survey on networked data streaming with Apache Kafka. *IEEE access* (2023).
[51] El Mehdi Saoudi and Said Jai-Andaloussi. 2021. A distributed Content-Based Video Retrieval system for large datasets. *Journal of Big Data* 8, 1 (2021), 87.
[52] Alan F Smeaton, Paul Over, and Aiden R Doherty. 2010. Video shot boundary detection: Seven years of TRECVid activity. *Computer Vision and Image Understanding* 114, 4 (2010), 411–418.
[53] Hui Sun, Weisong Shi, Xu Liang, and Ying Yu. 2019. VU: Edge computing-enabled video usefulness detection and its application in large-scale video surveillance systems. *IEEE Internet of Things Journal* 7, 2 (2019), 800–817.
[54] Vassilios Tsakanikas and Tasos Dagiuklas. 2022. A generic framework for deploying video analytic services on the edge. *IEEE Transactions on Cloud Computing* (2022).
[55] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *ACM SIGMOD'21*. 2614–2627.
[56] Ziyi Wang, Bo Lu, Yonghao Long, Fangxun Zhong, Tak-Hong Cheung, Qi Dou, and Yunhui Liu. 2022. AutoLaparo: A New Dataset of Integrated Multi-tasks for Image-guided Surgical Automation in Laparoscopic Hysterectomy. arXiv:2208.02049 [cs.CV]
[57] Weaviate. 2024. Vector Indexing. https://weaviate.io/developers/weaviate/concepts/vector-index#vector-cache-considerations.
[58] Hyeok Yoon and Ji-Hyeong Han. 2022. Content-based video retrieval with prototypes of deep features. *IEEE Access* 10 (2022), 30730–30742.
[59] Chih-Jui Yu et al. 2021. Lightweight Deep Neural Networks for Cholelithiasis and Cholecystitis Detection by Point-of-Care Ultrasound. *Computer Methods and Programs in Biomedicine* 211 (08 2021), 106382. https://doi.org/10.1016/j.cmpb.2021.106382
[60] Jinhui Yuan, Huiyi Wang, Lan Xiao, Wujie Zheng, Jianmin Li, Fuzong Lin, and Bo Zhang. 2007. A formal study of shot boundary detection. *IEEE Transactions on Circuits and Systems for Video Technology* 17, 2 (2007), 168–186.
[61] Xiao Zeng, Biyi Fang, Haichen Shen, and Mi Zhang. 2020. Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. In *ACM SENSYS'20*. 409–421.