Chat url: <https://chatgpt.com/share/6852dddf-f944-8012-ac66-5b1186801dc3>

**You said:**

I have an aws bucket called oriane-frames where I have folders (platform) and inside folders (video-code) and inside I have frames, for example: oriane-frames/instagram/jkknas32da/1.png...n.png

So now I want to get this frames and convert them to embeddings and store it in a database locally, for testing purposes. I know I can use OpenSearch database, which I already used and it is very expensive, also we got 300k$ credits in google cloud platform so I will have to set the pipeline there so i'm thinking what is the actual best way, even beyond industry current standards, to setup a embedding database to make massive search on video (video frames actually).

So the search types are:

1. search by image: if a user uploads an image it will be embeddings extracted and stored in our db, and then perform the search inside the db.

2. search by video: if a user uploads a video it will extract the frames and emebbedings and stored in our db, and then perform an image search per each frame. So it's the search 1. search by image but multiple times.

3. search by text: the user can prompt video-frames id's from our db by inputing a prompt like: 'dogs dancing under the rain', I think it's called semantic search, correct me if I'm wrong.

so here we need to decide which and why:

1. which MODEL we use to extract embeddings to get the most accurate results in the searches I mentioned earlier. Keep in mind that we are working in the search feature, which I know it's won't be perfect but we need it to filter most of the videos from our db before performing what we call 'Deep Vision' which basically is an SSCD model that the resulted search video frames one by one to find video-copies, or video remakes, etc. I'm explaining you this because our current SSCD pipeline takes a video codes array (['instagram.nadkjbhcss, tiktok.bhx97cnd, ...]), and fetches the frames from those videos and extracts the frames in-memory before infer the comparison. So with this new embeddings db search tool, we want to reduce the number of unnecesasary comparison by filtering with the embeddings search, so I'm wondering if the same extracted embeddings frames can be used in the Deep Vision inference, or we do need to extract them also with the pre-trained SSCD model?

2. which embeddings DATABASE we use. AND WHY? I heard about tensors db, vectors db, graph db, rag db, etc. A lot of terms but not sure which one is the best for our use case.

3. once we know which database, how we store the embeddings, how we INDEX them, if we relate them, if we need to add any metadata in the search for better retrieval. Basically, what is the best approach to store and organize our embeddings for our use case and why!

So you got the big picture, make a research and find the answers. I won't make any move if I don't understand why, so make sure to provide a clear answer for someone who has a basic education. I'm an software engineer, I care more about practice than theory, you don't need to get deep on math explanations, just be pragmatic, please.

As I told you we are in aws and expanding to gcp soon, so we want to be cloud agnostic, try finding open source tools, so we can deploy everywhere, while testing locally. However, if you find a good soluion which is not open source, do not avoid including it. We are looking for the best of the best for fast and optimal embedding retrival in a billions videos embeds db.

If you have any question that will help accomplish your mission before starting the deep research go on.

###### **ChatGPT said:**

Thanks for the detailed context—it’s very helpful. Before I begin the deep research, could you please clarify the following so I can tailor the recommendations more precisely:

1. Embedding Model Preference: Do you have any constraints or strong preferences (e.g., open-source only, must support multi-modal input like text and images)? Are you already using any model family (e.g., CLIP, BLIP, SAM, etc.) in other parts of your pipeline?
2. Latency vs Accuracy Trade-off: Are you optimizing more for search speed or accuracy, especially for the initial filtering before SSCD?
3. Query Volume and Scale: What’s the estimated number of queries per second (QPS) you expect the system to handle in production?
4. Hardware Availability: For local testing, are you running on CPU or do you have access to GPUs?
5. Metadata Requirements: Are you planning to use any metadata for retrieval (e.g., platform, upload time, user id) to improve filtering?

These answers will help narrow down the best model and database choices for your search pipeline across use cases.

##### **You said:**

1. Embedding Model Preference: No preferences. Multi-modoal would be nice, so we don't use 2 models, however if there is an image serach model very good but it is not multi-modal, I rather to have 2 different models than a less accurate one.

2. Latency vs Accuracy Trade-off: search must be fast, yes. but I undertsand it is something that we can parameter in the search query itself. Am I wrong? On the other hand, accuray it's very important too, so we just need to find the best ratio possible.

3. Query Volume and Scale: At the beggining few, we are opening our app to few clients first and then gradually scale. However, we expect high traffic in the next years, so keep scalability in mind, our video database is massive.

4. Hardware Availability: On cloud we will run elastic VM's with GPU. Locally I have 12 CPU's and 32RAM and 1 GPU with 8GB.

5. Metadata Requirements: Yes, basically we have 2 types of videos: public and private. Public comes from sources like instragram, tiktok, linkedin, etc. what we call platform in our folder organization. Inside the platform it will be folders with the video codes of that platform, and inside its video frames which the file name format is: nFrames\_secondWith2Decimals.png, i.e.: 1\_0.45.png, 2\_1.44.png, etc.

So I want to store: {

id: 'video uuid from our sql db',

platform: 'instagram',

video: 'video-code',

frame: 1, // frame number

second: 0.45, // frames second

create\_at: 'timemstamp of the entry'

}

This is just for 'public' videos. Private videos are comming directly from our app, it is the content (videos or images) that the user uploads. For that in our oriane-frames bucket we have the platform 'oriane' (our app) folder. But instead of having video codes folders inside it has user-id folders, our user's uuid from our sql db. So in this case I will store an entry to our videos table (same as with the public videos), so in this case the video code from our platform (oriane) will be a uuid as well. So the relations with the patform's users which the videos belong to will be handle in our sql sb. What do you think about this approach? Propose a better one if you can.

Last thing, rember always to explain why, no need to get deeper, it needs to get logical, that's it. As an engineer I rather simple systems the complex ones. I need it to be fast, perfomant, optimal, on the extraction and on the search side.

You can include a search strategies as well, which I forgot to mention. However, let's focus first on the extraction and storing, we will jump to the search strategies later. Let's go!

###### **ChatGPT said:**

Great. I’ll research the best end-to-end strategy for extracting video frame embeddings and storing them in a high-performance, scalable, cloud-agnostic embedding database. This will cover:

1. The best model(s) for extracting embeddings for video frame search (image, video, and text-to-image).
2. The best open-source or cloud-deployable vector databases for storing and indexing embeddings, with support for metadata.
3. Best practices for embedding storage, indexing, and metadata organization to ensure scalable, performant, and cost-efficient semantic and visual search.

I’ll prioritize solutions that are simple to set up, work well locally and in cloud environments, and support scaling to billions of embeddings. I’ll also evaluate whether your current metadata and folder organization is optimal or if improvements could simplify future development and retrieval.

I’ll get back to you soon with a practical, well-structured recommendation.

Research completed in 10m · 18 sources · 49 searches

Best Model and Database Strategy for Scalable Video Frame Embedding Search

# **Building a High‑Performance Semantic Video Frame Search System**

## **Overview**

To support image-to-image, video-to-video, and text-to-image searches across billions of video frames, we need a robust pipeline with three core components: (1) powerful image/text embedding models, (2) a scalable vector database for similarity search, and (3) a well-designed data schema (with metadata) and infrastructure for both public and user-specific content. Below we outline the recommended choices and architecture, focusing on practical trade-offs and engineering considerations.

## **Embedding Model Selection (Image & Text)**

Multi-Modal vs Separate Models: A single multimodal model that embeds both images and text into one semantic vector space is ideal for simplicity and cross-modal search. It avoids maintaining two disconnected embedding spaces (one for image-image similarity and another for text-image matching)

[jina.ai](https://jina.ai/news/jina-clip-v1-a-truly-multimodal-embeddings-model-for-text-and-image/#:~:text=tokens.%20,would%20be%20expected%20to%20support)

. OpenAI’s CLIP is the classic choice – it has an image encoder and text encoder trained jointly so that similar images and captions map to nearby vectors

[jina.ai](https://jina.ai/news/jina-clip-v1-a-truly-multimodal-embeddings-model-for-text-and-image/#:~:text=Language%E2%80%93Image%20Pretraining%29%20model,the%20image%2C%20and%20vice%20versa)

. CLIP enables both image↔image and text↔image searches out-of-the-box. However, CLIP’s original text encoder has limitations (only ~77 tokens input, and it was trained on short captions, making it less effective for longer or richer text queries)

[jina.ai](https://jina.ai/news/jina-clip-v1-a-truly-multimodal-embeddings-model-for-text-and-image/#:~:text=,kind%20of%20text%2C%20and%20do)

. Recommended Model – Enhanced CLIP Variants: Recent open-source CLIP variants improve accuracy across modalities. For example, Jina CLIP v1 (and v2) extend CLIP and achieve higher retrieval performance: Jina CLIP v1 improved text-only retrieval by ~165% and image-to-image retrieval by ~12% compared to OpenAI’s CLIP

[jina.ai](https://jina.ai/news/jina-clip-v1-a-truly-multimodal-embeddings-model-for-text-and-image/#:~:text=Jina%20CLIP%20v1%20%28jina,for%20working%20with%20multimodal%20inputs)

. It’s a *single* model for strong image embeddings and state-of-the-art text embeddings, eliminating the need for separate text models

[jina.ai](https://jina.ai/news/jina-clip-v1-a-truly-multimodal-embeddings-model-for-text-and-image/#:~:text=Jina%20AI%27s%20new%20multimodal%20embedding,for%20different%20modalities%20any%20more)

[jina.ai](https://jina.ai/news/jina-clip-v1-a-truly-multimodal-embeddings-model-for-text-and-image/#:~:text=this%20new%20model%2C%20users%20have,for%20working%20with%20multimodal%20inputs)

. This kind of model would let you handle text queries as effectively as a dedicated language model, while still aligning with the image space. Jina CLIP is available via HuggingFace and can be self-hosted, making it a pragmatic choice if you need better accuracy than vanilla CLIP. Another option is OpenCLIP (LAION’s open reproduction of CLIP) with a large ViT backbone (e.g. ViT-L/14) for stronger embeddings; these models are widely used for image search at scale (e.g. LAION’s 5B image index). They are not *quite* as advanced as JinaCLIP but are battle-tested and come in various sizes for speed/accuracy trade-offs. Separate Model Trade-offs: Using distinct models for different tasks (e.g. a specialized CNN for image similarity and a language model for text) is generally not necessary here. Separate models would complicate the system – you’d have to maintain two indices or convert text queries into the image embedding space via a secondary mechanism. If you did go that route, you might pick something like a ResNet50/ViT trained on image retrieval for image→image matching, and a BERT/SentenceTransformer for text queries. But then, mapping text to relevant images would require either training a bridging model or manually tagging concepts, which is far less robust than a joint model. In practice, modern multimodal models are strong enough that one model suffices for all three query types. For example, CLIP-based embeddings will retrieve semantically similar frames for both image and text queries (images with similar content have nearby CLIP vectors, and text queries will retrieve images whose content matches the text)

[elastic.co](https://www.elastic.co/search-labs/blog/openai-clip-alternatives#:~:text=Alternative%202%3A%20Cohere%20Image%20Embeddings,V3)

[elastic.co](https://www.elastic.co/search-labs/blog/openai-clip-alternatives#:~:text=Thanks%20to%20multimodality%2C%20we%20can,image%20similarity)

. In summary, use a single multimodal embedding model (such as CLIP or Jina CLIP) to generate 512–768 dimensional vectors for both frame images and text queries. This keeps the architecture simple and ensures all embeddings live in the same vector space for direct comparability. Performance Considerations: For real-time querying, you’ll want to optimize embedding extraction. Deploy the model on GPUs and batch-process inputs when possible. A ViT-based model can encode an image in tens of milliseconds on a GPU; batching multiple frames can amortize overhead. For video-to-video search, if a user provides a long video as a query, consider sampling a limited number of frames (e.g. 1 frame per second or one per scene cut) to represent the video – this will keep query latency reasonable. You can also explore techniques like model quantization to speed up inference (e.g. use FP16 or INT8 precision if supported)

[milvus.io](https://milvus.io/ai-quick-reference/how-are-embeddings-used-in-video-analytics#:~:text=Embeddings%20are%20stored%20in%20vector,performance%20without%20sacrificing%20critical%20details)

[milvus.io](https://milvus.io/ai-quick-reference/how-are-embeddings-used-in-video-analytics#:~:text=dimensionality%20%28e.g.%2C%20512,performance%20without%20sacrificing%20critical%20details)

. In practice, a ViT-B/32 CLIP model is a good starting point for local testing (quick and smaller), while a ViT-L/14 or Jina CLIP model can be used in production for higher accuracy once the system is scaled.

## **Vector Database for Scalable Similarity Search**

Storing and searching billions of embedding vectors requires a specialized vector database (vector index). Key requirements are: open-source or cloud-agnostic deployment, fast approximate nearest neighbor (ANN) search, ability to attach metadata (platform, video ID, timestamp, user, etc.), and horizontal scalability. Several mature options fit these needs:

* Milvus (Zilliz) – An open-source vector DB built for high scalability. Milvus can distribute indexes across many nodes and is explicitly designed to handle billion-scale vector corpora
* [milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=and%20simple%20API%20can%20save,generate%20embeddings%20on%20the%20fly)
* . It supports multiple ANN indexing algorithms (HNSW graphs, IVF flat or IVF with product quantization, etc.) for flexibility
* [milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=similarity%20to%20a%20reference%20image,scale%20AI%20prototyping)
* . Milvus tends to excel in raw performance and indexing speed (it can bulk-load vectors very quickly)
* [rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,index%2010%C3%97%20faster%20than%20some)
* [rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=embeddings%20,careful%20index%20selection%20and%20tuning)
* . The trade-off is that a full Milvus cluster is resource-intensive (the standalone “Milvus Lite” is lightweight for development, but the distributed cluster will consume significant CPU/RAM)
* [reddit.com](https://www.reddit.com/r/vectordatabase/comments/1dcvyrm/choosing_a_vector_db_for_100_million_pages_of/#:~:text=It%20seems%3A)
* . Milvus is a great choice if you anticipate *truly massive scale* and want fine control over index type and compression (for instance, Milvus can use IVF-PQ or even a disk-based index (DiskANN) to reduce memory usage at some cost to recall
* [rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,The%20ability%20to%20mix%20and)
* ). Its community and documentation are strong, and many large-scale projects use it in production
* [rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,hosted)
* .
* Qdrant – An open-source vector DB written in Rust, known for excellent performance and easy deployment. Qdrant uses the HNSW ANN algorithm under the hood and is highly optimized; benchmarks show top-tier throughput and low query latencies even with millions of vectors
* [rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=Qdrant)
* [rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=%2A%20Retrieval%20Speed%3A%20Qdrant%20%28open,Qdrant%E2%80%99s%20efficiency%20is%20attributed%20to)
* . A standout feature of Qdrant is its filtering with payloads – you can store arbitrary metadata with each vector and apply filters at query time with very little overhead
* [milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=%28e,scale%20AI%20prototyping)
* . This is perfect for our use case (filtering by video ID, user ID, etc.). Qdrant’s memory footprint is also efficient compared to some others. Currently, Qdrant is single-node (no built-in sharding in open source yet), but dynamic sharding is under active development
* [reddit.com](https://www.reddit.com/r/vectordatabase/comments/1dcvyrm/choosing_a_vector_db_for_100_million_pages_of/#:~:text=%E2%80%A2)
* . For billions of vectors, you would likely need to manually partition data across multiple Qdrant instances (or use Qdrant’s cloud which offers distributed support). Qdrant also recently introduced vector compression (product or binary quantization) to reduce memory usage
* [rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=Vector%20Databases%20for%20RAG%20Literature,fast%20SSDs%20to%20supplement%20RAM)
* . If your initial scale is in the low billions or less and can fit on a beefy server (or a few servers by partitioning, e.g. by video platform), Qdrant is a very pragmatic choice due to its simplicity and metadata filtering strength. Many teams use it successfully for production search where metadata filtering and performance are crucial
* [milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=%28e,scale%20AI%20prototyping)
* .
* Weaviate – Another popular open-source vector DB (in Go) with a lot of features. Weaviate uses HNSW for ANN and supports hybrid queries (vector search + keyword filters) and even has modules for on-the-fly embedding (though we won’t need that)
* [milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=similarity%20to%20a%20reference%20image,scale%20AI%20prototyping)
* . It supports sharding and replication in a cluster mode, and you can store metadata as “properties” on data objects. Weaviate’s appeal is its ease of use (it provides a GraphQL API, for example) and built-in support for things like scalar filtering and modular re-rankers. Performance-wise, Weaviate is comparable to others (some benchmarks show it slightly slower or faster depending on configuration, but in the same ballpark)
* [reddit.com](https://www.reddit.com/r/vectordatabase/comments/1dcvyrm/choosing_a_vector_db_for_100_million_pages_of/#:~:text=sharding)
* . One report noted Weaviate had very fast latency in one test (0.12s vs 0.9s for Milvus on a certain query)
* [reddit.com](https://www.reddit.com/r/vectordatabase/comments/1dcvyrm/choosing_a_vector_db_for_100_million_pages_of/#:~:text=sharding)
* , but results vary – it’s generally “fast enough” for interactive search (in the few tens of milliseconds for million-scale datasets on good hardware). It may also be somewhat lighter in resource usage, and it’s easy to deploy via Docker.

Other Options: If you prefer a managed service, Pinecone is a cloud vector DB known for seamless scalability, but it’s not open-source (and can become costly at scale). Since *cloud-agnostic* is a priority, an open solution is better. Elasticsearch/OpenSearch have vector-capabilities (KNN plugin) which could let you combine vector and text search, but they are not as optimized for billion-scale ANN. Postgres with pgvector extension is another route for smaller scale or if you need strict ACID properties – but vanilla pgvector won’t be as performant for billions (Timescale is working on pgvectorscale that dramatically improves Postgres ANN performance via disk-based indexes

[reddit.com](https://www.reddit.com/r/vectordatabase/comments/1dcvyrm/choosing_a_vector_db_for_100_million_pages_of/#:~:text=I%27ve%20been%20skeptical%20about%20Postgres,com%2Ftimescale%2Fpgvectorscale)

[reddit.com](https://www.reddit.com/r/vectordatabase/comments/1dcvyrm/choosing_a_vector_db_for_100_million_pages_of/#:~:text=Haven%27t%20tested%20it%20yet%2C%20but,and%20DX)

, but this is very new). For a pure ANN library approach, FAISS is excellent and could be used to build your own service (it’s the engine underlying many vector DBs). However, using a full-fledged vector DB (Milvus/Qdrant/Weaviate) will save you a lot of integration effort, since they handle metadata, persistence, and APIs for you. Recommendation: Both Milvus and Qdrant align well with the requirements. If you anticipate *needing to scale to distributed clusters quickly* and want the flexibility of multiple index types (for example, using HNSW at first, but switching to IVF-PQ or disk indexes as data grows), Milvus is a safe bet

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=and%20simple%20API%20can%20save,generate%20embeddings%20on%20the%20fly)

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,hosted)

. On the other hand, if your priority is simple deployment, lower memory usage, and powerful metadata filtering, Qdrant is an excellent choice

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=%28e,scale%20AI%20prototyping)

. You could even start with Qdrant on a single node (for development or up to a few hundred million vectors), and later move to a sharded Milvus setup if needed. Weaviate is equally capable, so if its feature set (GraphQL API or hybrid search) appeals to you, it’s a viable alternative. All three are open-source and can be self-hosted on any cloud or on-prem. The good news is that these databases have similar core concepts, so designing your system abstractly (with the ability to switch out the vector DB if necessary) is wise

[reddit.com](https://www.reddit.com/r/vectordatabase/comments/1dcvyrm/choosing_a_vector_db_for_100_million_pages_of/#:~:text=%E2%80%A2)

. In practice, you’ll choose one and optimize it, but avoid hard-coding to proprietary features so you remain cloud-agnostic and flexible. Index Type & Search Performance: Whichever DB you choose, you will need to configure the ANN index for your data. HNSW (Hierarchical Navigable Small World graphs) is a great default – it offers high recall (>95% typically) and low latency, at the cost of using a lot of memory for large datasets (because it stores links between many vectors in RAM)

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,filtering%20to%20control%20memory%20growth)

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=memory%20usage%20,filtering%20to%20control%20memory%20growth)

. If you truly have billions of frames (vectors), storing them all in memory may be infeasible. In that scenario, plan on using a compressed or disk-based index: for example, Milvus can do an IVF (inverted file) with Product Quantization which stores most of the vector data as compact codes on disk (drastically cutting memory needs, with some loss in precision)

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,The%20ability%20to%20mix%20and)

. Qdrant and Weaviate now also support Product Quantization (PQ) to compress vectors in memory

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,filtering%20to%20control%20memory%20growth)

. The idea is to reduce each 512-d vector to, say, 64 or 128 bytes instead of 2048 bytes (float32) by quantizing values. This yields a *huge* memory saving (8× or more) at a cost of a few percentage points of recall. You can also use disk ANN strategies – Milvus has a DiskANN/SPANN index for billion-scale datasets that keeps only graph or coarse structures in RAM and offloads vector data to SSD

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,The%20ability%20to%20mix%20and)

. These approaches let you scale to billions of vectors without needing petabytes of RAM. The trade-off is slightly slower query times and complexity in tuning parameters (e.g. number of IVF centroids, HNSW ef search depth, etc.). For fast, real-time queries, ensure you tune the index parameters. For HNSW, you’ll adjust M (graph connectivity) and ef (search breadth) to balance speed vs recall. For IVF, you’ll adjust number of clusters and PQ bytes to balance recall vs memory. In testing, you might find that a hybrid approach works: e.g. use HNSW for a “hot” subset of data (recent or frequently queried frames) and IVF-PQ for the full corpus in a tiered setup – or simply rely on caching results for popular queries. Since *precision in retrieval* is important (e.g. you want the system to actually find that “red car turning left” scene or an exact frame match), lean towards high-recall configurations. As one expert note: graph-based indexes (HNSW) typically achieve higher recall than heavy compression indexes, at the cost of memory

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,off%20as%20needed)

. You can mitigate memory costs with moderate compression (e.g. use 8-byte PQ code *plus* keep a smaller HNSW graph in RAM). The bottom line: expect to experiment with index settings on your data to meet latency/accuracy targets. Fortunately, these vector DBs allow re-indexing or hybrid index usage fairly flexibly.

## **Storing Embeddings and Metadata**

Each video frame embedding needs to be stored along with metadata like platform, video ID (or code), timestamp/frame number, user ID (owner), and any other context (e.g. frame extraction date). Best practice is to store this metadata *inside the vector database* if the DB supports it, so that you can filter and retrieve it in one query. All the recommended vector DBs allow this:

* In Qdrant, you can store a JSON payload with each vector (e.g. {"platform": "YouTube", "video": "abc123", "frame\_time": 42.0, "user\_id": null, "created\_at": "2025-06-18T..."} ). Qdrant’s queries can use these fields for filtering, like filter: platform="YouTube" AND user\_id = 123 to restrict results
* [qdrant.tech](https://qdrant.tech/documentation/guides/multiple-partitions/#:~:text=2,filter%20vectors%20for%20each%20user)
* [qdrant.tech](https://qdrant.tech/documentation/guides/multiple-partitions/#:~:text=,%7D%20%7D)
* . Qdrant emphasizes this payload filtering capability as a core feature
* [milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=%28e,scale%20AI%20prototyping)
* .
* Weaviate uses an object schema: you might define a class Frame with properties platform (string), videoID (string), timeSec (number), user\_id (int), etc., and a vector. You can then do GraphQL queries filtering by those fields. This is very straightforward and Weaviate handles the indexing behind the scenes (it can build an inverted index for the metadata properties to speed up filtering).
* Milvus (2.x) allows attaching scalar fields to vectors in a collection. You can either store an integer ID that links to an external relational DB, or directly store fields in the collection schema. Milvus’s querying for metadata is a bit less flexible out-of-the-box than Qdrant/Weaviate (you often retrieve by ID then join metadata), but recent versions do support scalar filters in queries. Alternatively, you can use Milvus in conjunction with an external database (store the vector ID and metadata in PostgreSQL, for example). However, keeping it all in one system is simpler.

Schema Design: Define a clear schema for the metadata. For example, a composite ID might be useful: you could concatenate platform:videoID:timestamp into a single unique ID string for each frame, which becomes the primary key in the vector DB. This unique ID can also serve as the filename or S3 key for the image (and indeed your S3 structure is platform/video-code/frame.png). Storing this ID and fields like platform, video\_code, and frame\_time separately can help with filtering (e.g. filter by platform or by video). For public vs private videos, include a user\_id or owner field. For public content, this might be user\_id = null or a special value indicating public domain. For user-uploaded videos, set user\_id to the owner’s ID. It’s also useful to have a boolean flag like is\_public = true/false in case you want to quickly query “only public” content, but this can be derived from user\_id being null or not. Additionally, you might store a thumbnail URL or S3 path in the metadata (or you can reconstruct it from the ID when needed). For instance, storing s3\_path: "s3://oriane-frames/YouTube/abc123/00042.png" for quick access to the frame image. However, duplicating the S3 path might be unnecessary if you can build it from platform/video/time metadata. When indexing billions of frames, consider batch insertion and index building procedures. It will be far more efficient to bulk-insert vectors (hundreds or thousands at a time) than one by one. All these databases have a bulk import API. After ingest, the DB will build or update its ANN index. For large static datasets, you might even use an offline indexing tool (Milvus provides a standalone index builder for IVF, etc.). Pay attention to index build time and plan for how to update indexes if new data is continuously added (perhaps schedule periodic index rebuilds or use the DB’s incremental indexing if supported). Dimensionality & Storage: The chosen embedding model dictates vector dimensionality (e.g. CLIP ViT-L/14 gives 768-d vectors). These are typically stored as 32-bit floats by default. That’s 3 KB per vector (768 \* 4 bytes). At 1 billion vectors, that’s ~3 TB just in raw vector data – clearly too high for RAM. This is why we stress compression or quantization. You can instruct the DB to compress vectors (e.g. store as INT8 or apply PQ) to shrink this. It’s a trade-off: 8-bit quantization could cut memory 4× with minimal recall drop if done well

[medium.com](https://medium.com/@benitomartin/balancing-accuracy-and-speed-with-qdrant-hyperparameters-hydrid-search-and-semantic-caching-part-84b26037e594#:~:text=Balancing%20Accuracy%20and%20Speed%20with,Scalar)

. Also, you might not need *all* 768 dimensions for effective retrieval – some practitioners reduce vectors to 256 or 384 dims via PCA/Autoencoder if they can tolerate some loss, but given modern models, it’s usually better to keep full dimensions and use quantization rather than manual reduction. In any case, plan your hardware around the vector storage needs: a combination of RAM and fast SSD (NVMe) is ideal. Many vector DBs (like Milvus, Weaviate) will cache index graph in RAM but keep vector data on disk if using disk indexes

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,The%20ability%20to%20mix%20and)

, so fast SSDs and high IOPS will be important. Indexing Strategy: Organize your data in the DB in a way that makes sense for your queries. For example, you could create separate collections or indexes per platform (YouTube frames in one, Vimeo in another, etc.) if queries are usually platform-specific. But if you want the ability to search globally across all platforms, one big index is simpler. Given the scale, one big index is fine as long as you have the metadata to filter if needed. Some vector DBs also support partitions or segments internally – e.g. Milvus can partition by a tag (like year or platform) and only search within certain partitions for efficiency. That could be useful if, say, 90% of queries target recent content so you partition by date. This is an advanced optimization; initially, focus on just getting everything into one place and working, and measure performance.

## **Handling Public vs Private Content (Multi-Tenancy)**

Your system will store both public video frames and private user-uploaded video frames. It’s critical to isolate private data so that users only search their own videos (plus whatever public content you allow them to search). There are two main approaches: (1) *separate indexes* for different data scopes, or (2) *shared index with metadata filtering*. Single Index with Metadata Filtering: The simpler and more scalable approach is to keep all frames in one big index, but use the user\_id (tenant identifier) in metadata to filter results per user. At query time, your application will add a filter like user\_id = current\_user OR is\_public = true so that a user searches over public content plus their own content only. This “virtual multi-tenancy” approach is recommended by vector DB best practices

[pinecone.io](https://www.pinecone.io/learn/series/vector-databases-in-production-for-busy-engineers/vector-database-multi-tenancy/#:~:text=In%20this%20pattern%2C%20you%20would,relevant%20tenant%E2%80%93%20user%2C%20data%20source)

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-handle-multitenancy-in-semantic-search-applications#:~:text=partition%20data%20at%20the%20storage,specific)

. For example, Pinecone suggests using a single index with tenant metadata and applying a filter for the tenant on each query

[pinecone.io](https://www.pinecone.io/learn/series/vector-databases-in-production-for-busy-engineers/vector-database-multi-tenancy/#:~:text=In%20this%20pattern%2C%20you%20would,relevant%20tenant%E2%80%93%20user%2C%20data%20source)

. Qdrant’s documentation likewise shows filtering by a group\_id or user\_id field on queries to implement multi-tenant search

[qdrant.tech](https://qdrant.tech/documentation/guides/multiple-partitions/#:~:text=2,filter%20vectors%20for%20each%20user)

. The advantage here is that you *can* easily allow cross-tenant queries when needed (e.g. an admin searching across all public + private data), and you only manage one index. The vector DB will efficiently handle the filter by checking metadata of candidates – this typically adds negligible overhead if the index is well-organized for it

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=%28e,scale%20AI%20prototyping)

. Multiple Indexes (Namespaces or Collections): In some cases, you might isolate data by creating separate indices or namespaces per user or per group of users. For instance, Pinecone supports namespaces which physically separate vectors by tenant, ensuring queries only hit one tenant’s data

[pinecone.io](https://www.pinecone.io/learn/series/vector-databases-in-production-for-busy-engineers/vector-database-multi-tenancy/#:~:text=Data%20isolation%20and%20efficient%20querying,with%20namespaces)

[pinecone.io](https://www.pinecone.io/learn/series/vector-databases-in-production-for-busy-engineers/vector-database-multi-tenancy/#:~:text=Any%20query%20only%20interacts%20with,need%20to%20delete%20the%20namespace)

. Milvus or Weaviate could use separate collections/classes for each user as well. However, with potentially millions of users, having separate indexes for each isn’t practical (too many indices to manage, and you lose the ability to do combined searches easily). A compromise could be to maintain two top-level indexes: one for public content and one for all private content. Then on a user query, you’d query both indexes (public index and their user-specific index) and merge results. But merging ANN results from two indexes is non-trivial to get right in terms of scoring, and this approach doesn’t scale well if every user has their own index. It’s usually simpler to stick with one index and use filtering. As a safeguard, you can enforce at the application level that every query must include the appropriate user\_id filter (except for purely public searches), to avoid any leakage

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-handle-multitenancy-in-semantic-search-applications#:~:text=identifier%20%28e.g.%2C%20,specific)

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-handle-multitenancy-in-semantic-search-applications#:~:text=index%20aliases%20or%20routing%20to,For%20access%20control%2C%20combine)

. Metadata schema for tenants: Use a field like tenant\_id or user\_id on each vector

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-handle-multitenancy-in-semantic-search-applications#:~:text=identifier%20%28e.g.%2C%20,specific)

. Public frames can have user\_id = null or perhaps user\_id = 0 and is\_public=true. Then:

* For a public search (e.g. an open search endpoint for all users), filter is\_public = true (or user\_id = null).
* For a user’s private search, filter user\_id = {their ID} (and possibly OR is\_public = true if you want to include public results too). If the DB doesn’t support OR in filters natively, you may have to run two queries (one for user’s content, one for public) and combine, or tag combined content as accessible. Some DBs allow an OR in filter (Qdrant’s filter supports boolean logic, for example), but if not, you could duplicate public content under every user in metadata (not ideal), so better use a DB that supports the needed filter logic or do two searches.

Data Isolation & Security: By using the filter approach, you ensure no user ever sees another’s private frames, as long as the filter is always applied. It’s wise to implement this at a middleware level in your API – e.g. wrap the vector search call so it automatically adds user\_id = current\_user to every query. This avoids mistakes where someone forgets to filter. For administration, you can have special roles that can search without that filter (or across specific subsets). If you have very large tenants (say one user has an enormous collection of videos), you might in the future break out that user’s data into their own index for performance reasons

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-handle-multitenancy-in-semantic-search-applications#:~:text=Performance%20and%20cost%20are%20critical,specific%20results)

. But until that complexity is needed, stick with shared index + metadata, which is much easier to manage. File Organization (S3) for Public vs Private: On the storage layer (S3), it’s prudent to separate private videos’ frames from public ones, mostly for access control and manageability. One strategy is to use different S3 prefixes or buckets. For example:

* s3://oriane-frames/public/{platform}/{video-id}/{frame}.png for public video frames (the existing structure).
* s3://oriane-frames/private/{user-id}/{video-id}/{frame}.png for user uploads, or even a separate bucket oriane-user-frames/{user-id}/....

This way, you could apply S3 bucket policies or cloud IAM rules to ensure only authorized access to private content. Even if all frames ultimately feed into the same search index, keeping them organized by visibility helps avoid mistakes. It also simplifies housekeeping (e.g. if a user deletes a video, you can easily find all frames for that video in their folder to delete). Include the storage location or at least the necessary keys in your metadata so that when search results come out, you can present or fetch the corresponding frame. Typically, the search result might return something like {video: abc123, frame\_time: 42.0, platform: YouTube} – from that, your application can construct a URL to the image (https://.../YouTube/abc123/42.0.png) or query a database of video metadata to get more info (like the video title, etc. if needed for display).

## **System Architecture and Deployment**

*Example of a semantic search architecture (using AWS services in this diagram): video frames are processed through an embedding model to generate vectors stored in a vector database; queries are encoded by the same model and the vector DB returns similar content.* In our scenario, the core components include an embedding service, a vector index service, and the application logic for querying and results. Below is a simple but performant architecture outline:

* Data Ingestion Pipeline: A backend service (or batch job) is responsible for processing new videos. For each video, extract frames (which you’ve done, stored in S3). Then compute embeddings for each frame using the chosen model. This embedding step can be parallelized – e.g. use AWS Batch or Kubernetes jobs to run through videos frame-by-frame on GPU workers. Given the volume, you should batch inserts to the vector DB (e.g. accumulate 1,000 embeddings then call an upsert API). Monitor this pipeline’s throughput; you may need multiple workers to keep up with new content. For existing datasets, do a one-time bulk indexing (leverage any bulk import utilities of your DB to speed this up). After indexing, the vector DB will build its ANN structures (if not done incrementally). You might maintain a cache or database of processed videos (so you know which frames/embeddings are already indexed).
* Embedding Service (online queries): For search queries, you need to embed the query in real time. Set up a microservice or function for this. For image queries, the user will upload an image or provide a frame – the service loads the embedding model (likely running on a GPU server) and returns the vector. For video queries, the service should sample and embed a handful of frames from the video file (e.g. if a user uploads a clip, you might take 1 frame per second or the middle frames of the video). It’s important to constrain how many frames to embed from a query video for latency reasons (maybe limit to the first N seconds or use key frames). For text queries, simply embed the text with the text encoder. This embedding service can be part of your main API server if that server has GPU access, or it can be separate (for example, an AWS SageMaker endpoint or a Docker container with PyTorch that your API calls). Aim to keep the query embedding latency low (perhaps ~50ms for an image on a GPU). If using a large model like Jina CLIP v2 (0.9B params), you may need to use a GPU with sufficient memory and possibly run inference in FP16. For scaling, you can run multiple replicas of this service behind a load balancer to handle concurrent queries.
* Vector Search Service: The vector database (Milvus/Qdrant/etc.) will likely run as its own service or cluster. For development, you could run it on a single machine (even your local machine via Docker). For production with billions of vectors, you’ll deploy it on a cluster of machines or high-memory instances. For example, Milvus can be deployed on Kubernetes with multiple query nodes and index nodes; Qdrant you might run one instance per machine and handle sharding at the app level until their clustering is out. Either way, it’s a network service your application will query (via REST, gRPC, or a client library). Ensure this service is configured to use disk appropriately if the dataset doesn’t fit fully in RAM (fast NVMe disks, etc.). Also, monitor its resource usage (CPU for query time, memory for index).
* Application API Layer: This is the part that glues everything together. When a user makes a search request (whether via a UI or another service), your application will:
  1. Authenticate/authorize the user (if needed), then
  2. Embed the query using the embedding service (if the query is an image or text; if it’s a video, embed multiple frames).
  3. Query the vector DB: Formulate a vector similarity query. If multiple query vectors (e.g. from a video’s frames), you have a couple of options: (a) run multiple searches (one per frame) and merge the results, or (b) combine the frame vectors into one (e.g. average or pick the medoid) and do a single search. Merging results can be as simple as taking the top K from each frame query and then re-sorting them by similarity score, or more sophisticated like learning a fusion function – but initially, a union of results sorted by score is fine. Many similar frames from the same video might appear; you may want to group results by video when presenting (so you don’t show 10 frames from one video in the top 10). That can be done by post-processing the metadata (group by video ID).
  4. Apply Metadata Filters: Always include the appropriate filters in the vector DB query. For example, if the user searches within their private videos (say searching “dog in my videos”), add user\_id = current\_user. If they’re searching globally, maybe is\_public = true (or if you allow a combined search of public + their content, then you use the OR logic as discussed). Most vector DBs allow adding filter conditions directly in the similarity query call
  5. [qdrant.tech](https://qdrant.tech/documentation/guides/multiple-partitions/#:~:text=,%7D%20%7D)
  6. , so the results come back already filtered.
  7. Return Results: The DB will return the nearest neighbors – typically you get a list of IDs with similarity scores and maybe the stored metadata. Your app can then fetch the actual frame images or generate thumbnail URLs from the metadata. For each result, you might want to include additional info – e.g. the video title (which means your system should have a way to map video\_id to a video title/description from the platform’s API or your database). At minimum, you’ll return the platform, video ID and timestamp so that the user can locate that frame (possibly even link to the original video at that timestamp). For example, if platform is YouTube, a link to https://youtube.com/watch?v=VIDEO\_ID&t=42s could be provided.
* System Architecture Simplicity: The above can be implemented with a few components: a web API (could be a Node/Express or Python FastAPI, etc.) that calls a vector DB client and an embedding model. To scale, you’d containerize the vector DB (if single node) or run the DB cluster on Kubernetes. The embedding model might run on a dedicated GPU instance (container) and the API calls it via HTTP or RPC. This separation allows you to scale them independently – e.g. if embedding becomes the bottleneck, add another GPU worker; if DB queries are slow, scale out the DB or optimize the index. Cloud-agnostic deployment means you might use Terraform or Kubernetes to be able to deploy on AWS, GCP, etc., without relying on proprietary services. For instance, you could deploy Milvus on AWS EKS or GKE, and run a custom embedding service on an EC2 VM or a GKE node with GPU.
* Performance Tuning: Aim for low-latency responses by doing as much in parallel as possible. If you have multiple query vectors (video query scenario), consider issuing parallel searches to the DB (threads or async calls) and then merging, rather than sequential. Also, use the fact that vector DBs can return more results than needed and then filter – e.g. if a filter is hard to index, the DB might retrieve top 100 neighbors then apply the filter. Adjust the number of results (limit) and filtering strategy to ensure accuracy. For text queries, the embedding model might actually be slower than the DB lookup, so optimizing the model is key (e.g. smaller model or running on GPU). For image queries, the DB search (especially if using HNSW) might dominate if the dataset is huge but can still typically be under 100ms for billions if the index is well-structured and you have enough CPU shards. If you find queries are slow, you can tune ANN parameters (higher ef for recall can slow down queries – you might set a slightly lower ef if you can tolerate a tiny recall drop, or scale out to more CPU cores to compensate).
* Testing and Iteration: For local testing, you can start with a subset of your data (say 100k frame embeddings) and run a local vector DB instance. Use a smaller model like CLIP ViT-B/32 to verify the end-to-end pipeline (embedding -> search -> result). This will let you validate the relevance of results (qualitatively) and measure baseline latency. It’s also easier to debug with small data. Ensure your metadata filtering works correctly by simulating a couple of users and public data. You can test on a single machine (even without GPU by using a CPU embedding model variant, though slower). For the vector DB, both Qdrant and Milvus have Docker images – you can spin one up and use their Python clients to insert/query data. Once the local prototype is working, you can gradually scale up – try 1 million vectors in the DB and see how query time and memory look. This may involve moving to a bigger instance or enabling ANN indexes (for initial small tests you could even use brute-force search to verify correctness).
* Cloud Deployment (Scaling up): When moving to production scale, containerize each service: e.g. embed-service (with model), search-service (API server that also contacts vector DB), and the vector-db itself. Use orchestration (Kubernetes, Docker Compose, etc.) to deploy. In AWS, you might use EKS for Kubernetes or ECS for containers. Ensure that the vector DB has persistence (if using local disks, attach EBS volumes or use a StatefulSet for Milvus so data isn’t lost on pod restarts). Also set up monitoring for the vector DB – it’s important to track query latency, CPU usage, memory, index build times, etc. Tuning those will be an ongoing engineering task.
* Metadata and Schema Management: Maintain a small relational database or at least a config file for things like platform codes, user info, etc., if needed for the app. For example, you might have a table of videos (video ID → title, duration, etc.) which you can join when displaying search results. Also log the queries and results for analysis (to see if the embedding model is working well or if you need to fine-tune it on your domain).
* Optimizing for Throughput: If you expect frequent queries, note that vector DBs can handle a good number of QPS but heavy image/video queries will also load the embedding service. You could implement a simple cache for query embeddings – e.g. if the same image is queried often, cache its vector (perhaps keyed by an image hash). Similarly for text queries: caching the embedding of common text queries (like “cat” or “car crash video”) would save recomputation. This is an optimization to consider if you hit scale. Also, for video queries, perhaps cache the result of video-to-frames-to-embeddings if users might submit the same video repeatedly.
* Pragmatic Trade-offs: Focus on what brings the biggest benefit first: using a *good embedding model* and a *reliable vector index* are top priorities. Fine-tuning the model or the index comes next. Don’t over-engineer rare scenarios (e.g. searching *across* users might not be needed if each user only cares about their own and public data). Keep the architecture modular – e.g., if later you find a new model that’s better, you can swap out the embedding service; if you outgrow Qdrant on one node, you can spin up a Milvus cluster and re-index in the background, then switch queries to that. Abstracting the vector search behind a repository interface in your code can make such changes easier
* [reddit.com](https://www.reddit.com/r/vectordatabase/comments/1dcvyrm/choosing_a_vector_db_for_100_million_pages_of/#:~:text=%E2%80%A2)
* .

By following these recommendations, you’ll have a semantic search system that is *both* high-performing and practical to maintain. You’ll use state-of-the-art embeddings to capture visual and textual semantics, an open-source vector database to provide millisecond-level similarity search (even with billions of items)

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,hosted)

, and a clean metadata scheme to handle multi-user data governance. This design is cloud-agnostic and can be deployed on any infrastructure (from your laptop for dev, to an AWS/GCP cluster for production). Start simple – perhaps index a few million frames and get the search working – then iterate: measure latency, tweak the index settings, possibly upgrade hardware or distribute the load as needed. With careful engineering, you can achieve a system where a user can input an image, video, or text and in a fraction of a second retrieve the most semantically similar video frames from a sea of billions. Sources: The recommendations above are grounded in current industry practices and expert insights. For instance, multimodal models like CLIP and Jina CLIP are known to excel at cross-modal retrieval

[jina.ai](https://jina.ai/news/jina-clip-v1-a-truly-multimodal-embeddings-model-for-text-and-image/#:~:text=Jina%20CLIP%20v1%20%28jina,for%20working%20with%20multimodal%20inputs)

[elastic.co](https://www.elastic.co/search-labs/blog/openai-clip-alternatives#:~:text=Compared%20to%20the%20original%20OpenAI,see%20in%20the%20chart%20below)

, and open-source vector databases such as Milvus, Qdrant, and Weaviate are explicitly built to handle billion-scale vector search with filters

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=and%20simple%20API%20can%20save,generate%20embeddings%20on%20the%20fly)

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-choose-between-pinecone-weaviate-milvus-and-other-vector-databases#:~:text=%28e,scale%20AI%20prototyping)

. The importance of metadata-driven multi-tenancy is emphasized in both Qdrant and Milvus communities

[qdrant.tech](https://qdrant.tech/documentation/guides/multiple-partitions/#:~:text=2,filter%20vectors%20for%20each%20user)

[milvus.io](https://milvus.io/ai-quick-reference/how-do-i-handle-multitenancy-in-semantic-search-applications#:~:text=identifier%20%28e.g.%2C%20,specific)

. Balancing embedding dimensionality and using techniques like product quantization or DiskANN is a common strategy to scale vector search while preserving accuracy

[milvus.io](https://milvus.io/ai-quick-reference/how-are-embeddings-used-in-video-analytics#:~:text=fine,performance%20without%20sacrificing%20critical%20details)

[rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-for-rag-literature#:~:text=,The%20ability%20to%20mix%20and)

. The architecture outlined follows patterns used in semantic search systems (as illustrated in AWS’s OpenSearch semantic search example

[aws.amazon.com](https://aws.amazon.com/blogs/big-data/amazon-opensearch-services-vector-database-capabilities-explained/#:~:text=Semantic%20search%20is%20able%20to,Service%20as%20the%20vector%20database)

). By combining these pragmatic techniques, your system will leverage the best of modern AI (embeddings) and best of databases (fast ANN with filtering) to meet the requirements. Good luck with your implementation!

##### **You said:**

thanks!

1. Embedding Model Selection (Image & Text): I want to try JinaCLIP and then OpenCLIP.

2. Vector Database for Scalable Similarity Search: I heard about those 3, I think I'm gonna go with Qdrant. However, I would like you to make a comparision table of features betweent hese 3 vec dbs.

3. Storing Embeddings and Metadata: we will get deep on this point once we setup the vec db.

4. Handling Public vs Private Content (Multi-Tenancy): Forget this, do not change my s3 structure. private files are inside oriane's platform folder, period. no need to make public or private folder, even the data is 'public' from known platforms our buclets are all privates, our backend handles who sees what.

5. System Architecture and Deployment: you mention a grapg but I don't see any diagram, don't worry. I do not need it. We will jump to architecture after testing and validating it locally.

Let's finish deciding which db and then we will jump to code!

###### **ChatGPT said:**

Thought for 53 seconds

Below is a concise feature-by-feature snapshot of the three leading open-source vector databases you asked about. I focused on capabilities that matter most for your use-case (billions of image/video-frame vectors, heavy metadata filtering, cloud-agnostic deployment, eventual scale-out).

| **Feature / Cap** | **Qdrant** | **Milvus (2.x)** | **Weaviate** |
| --- | --- | --- | --- |
| License & Core Lang | Apache-2.0 • Rust core  [github.com](https://github.com/qdrant/qdrant)  [github.com](https://github.com/qdrant/qdrant?utm_source=chatgpt.com) | Apache-2.0 • Go/C++ core  [github.com](https://github.com/milvus-io/milvus?utm_source=chatgpt.com)  [github.com](https://github.com/milvus-io/milvus?utm_source=chatgpt.com) | Apache-2.0 • Go core  [github.com](https://github.com/weaviate/weaviate?utm_source=chatgpt.com)  [weaviate.io](https://weaviate.io/blog/scaling-and-weaviate) |
| Primary APIs / Clients | REST & gRPC; Python, JS, Go, Rust, Java SDKs | gRPC & REST; Python, Java, Go, Node, C# | GraphQL and REST; Python, JS/TS, Go, Java |
| Index Algorithms | HNSW (filter-aware), Flat; PQ & Binary quantization on top  [cohorte.co](https://www.cohorte.co/blog/a-developers-friendly-guide-to-qdrant-vector-database?utm_source=chatgpt.com)  [qdrant.tech](https://qdrant.tech/documentation/guides/quantization/?utm_source=chatgpt.com) | HNSW, IVF-Flat, IVF-PQ, SCANN, DiskANN (SSD)  [milvus.io](https://milvus.io/blog/diskann-explained.md?utm_source=chatgpt.com)  [milvus.io](https://milvus.io/docs/index-explained.md?utm_source=chatgpt.com) | HNSW (per-shard), Flat; (PQ plug-in in beta)  [weaviate.io](https://weaviate.io/blog/scaling-and-weaviate)  [weaviate.io](https://weaviate.io/developers/weaviate/search/hybrid?utm_source=chatgpt.com) |
| Compression / Memory-saving | Scalar, Product & Binary Quantization (up to 97 % RAM cut)  [github.com](https://github.com/qdrant/qdrant?utm_source=chatgpt.com)  [qdrant.tech](https://qdrant.tech/articles/binary-quantization/?utm_source=chatgpt.com) | IVF-PQ, Scalar Quant., DiskANN keeps full vectors on SSD with PQ codes in RAM  [milvus.io](https://milvus.io/blog/diskann-explained.md?utm_source=chatgpt.com)  [milvus.io](https://milvus.io/docs/index-explained.md?utm_source=chatgpt.com) | Optional PQ (early) + shard-level mmap; relies mainly on RAM for HNSW |
| Disk-based / Tiered Storage | Vectors on disk with quantized RAM-resident HNSW; automatic WAL & snapshots | DiskANN / SPANN native SSD index; tiered cold storage via MinIO/S3 | Persistent WAL; HNSW index per shard on disk; no dedicated DiskANN equivalent |
| Metadata Filtering | Payload JSON; boolean logic AND/OR; low-overhead “filterable HNSW”  [cohorte.co](https://www.cohorte.co/blog/a-developers-friendly-guide-to-qdrant-vector-database?utm_source=chatgpt.com) | Scalar fields & partitions; filter at query time (less expressive than Qdrant) | First-class typed properties; GraphQL filters; hybrid BM25 + vector in one query  [weaviate.io](https://weaviate.io/developers/weaviate/search/hybrid?utm_source=chatgpt.com)  [rohan-paul.com](https://www.rohan-paul.com/p/vector-databases-in-document-retrieval?utm_source=chatgpt.com) |
| Hybrid / Keyword + Vector | Keyword filter + vector similarity (no BM25 scoring) | Requires external text engine (OpenSearch, etc.) for keyword fusion | Native hybrid (BM25 × vector fusion) built-in  [weaviate.io](https://weaviate.io/developers/weaviate/search/hybrid?utm_source=chatgpt.com) |
| Sharding / Replication | Single-node today; dynamic sharding in active dev; replication in cloud edition  [qdrant.tech](https://qdrant.tech/) | Mature cluster: etcd + Pulsar; auto-shard, replica factor; strong / eventual consistency  [en.wikipedia.org](https://en.wikipedia.org/wiki/Milvus_%28vector_database%29?utm_source=chatgpt.com) | Built-in shard & Raft metadata replication; tunable stateless nodes; zero-downtime rolling updates  [weaviate.io](https://weaviate.io/blog/scaling-and-weaviate) |
| GPU Acceleration | CPU-only search; GPU on roadmap | CUDA/RAFT for IVF-PQ & graph build; 10×+ index build speed  [en.wikipedia.org](https://en.wikipedia.org/wiki/Milvus_%28vector_database%29?utm_source=chatgpt.com) | CPU-only search; GPU offload via external plug-in |
| Multi-tenancy Patterns | Single index + JSON filters recommended; namespaces in cloud | Databases & collections; RBAC; collection-level partitions | Namespaces (multi-DB) + per-property ACLs; GraphQL auth filters |
| Managed Cloud Option | Qdrant Cloud, Hybrid Cloud  [qdrant.tech](https://qdrant.tech/) | Zilliz Cloud | Weaviate Cloud (serverless & BYO-cloud) |
| Local Dev Experience | One-liner Docker; runs in <1 GB RAM with quantization | milvus-lite (embedded) or Docker; needs etcd & pulsar even in dev | Single binary Docker; auto-schema, hot-reload |

### **How this maps to your needs**

| **Requirement** | **Best Fit** |
| --- | --- |
| Billions of image vectors, want tight metadata filters | Qdrant: lean Rust core keeps latency low; filter-aware HNSW avoids “retrieve ⇢ post-filter” penalty. Quantization cuts RAM drastically – useful when every frame becomes a row. |
| Need disk-based index when RAM is not enough | Milvus: offers true DiskANN & GPU-accelerated IVF-PQ, giving you a predictable path when data outgrows RAM. |
| Hybrid keyword + vector search (if later needed) | Weaviate shines here with built-in BM25 fusion. |
| Cluster maturity today | Milvus > Weaviate ≈ Qdrant (Qdrant single-node OSS, distributed in cloud/roadmap). |
| Quick local PoC | All three give a Docker image; Qdrant’s single-binary footprint is the smallest. |

### **Pragmatic recommendation**

Because you value simple deployment, strong metadata filtering, and immediate local testing, go ahead with Qdrant for the first iteration:

1. Spin up Qdrant via Docker on your workstation (it will run fine on your 32 GB RAM box).
2. Index a pilot set of frame embeddings (e.g., 1 M frames) and test search latency/recall.
3. Enable scalar/product quantization once vectors exceed RAM.
4. When you move to GPU servers, embedding extraction remains separate; Qdrant doesn’t need GPU.

Keep an eye on Qdrant’s upcoming clustered open-source mode. If, in production, you hit a ceiling before that lands, you can migrate to Milvus (similar API concepts, Python clients for both). The table above should make that future switch predictable.

Let me know when you’re ready, and I’ll walk you through:

* installing Qdrant locally,
* loading the first batch of Jina CLIP embeddings,
* and writing the minimal search code (image / video / text queries).

Favicon

Favicon

Favicon

Sources

##### **You said:**

you made me more confused, lol. make a choice for our use case and defend it.

###### **ChatGPT said:**

Thought for 52 seconds

### **My pick: Qdrant – start-to-finish**

| **Why it matters for your pipeline** | **Qdrant** | **How it compares** |
| --- | --- | --- |
| Fast ANN + metadata filter in one pass  Image/text/video queries always come with filters (platform, video\_id, user\_id, …). Qdrant’s HNSW is *filter-aware*: it prunes candidates during graph traversal instead of “search first, post-filter later”, so latency doesn’t blow up on highly selective queries. | Built-in payload JSON, Boolean AND/OR logic; planner switches strategy automatically when a filter is very selective  [github.com](https://github.com/orgs/qdrant/discussions/322?utm_source=chatgpt.com) | Milvus supports scalar filters, but they run *after* vector search; more RAM burned per query to hold intermediate bit-sets  [stackoverflow.com](https://stackoverflow.com/questions/78939051/how-milvus-work-when-search-with-expr-and-vector?utm_source=chatgpt.com) |
| Memory efficiency – billions of 768-d CLIP vectors will not fit in RAM raw. | Product + binary quantization cut RAM up to -97 % with single-byte codes; originals stay on SSD for re-check if needed  [qdrant.tech](https://qdrant.tech/documentation/guides/quantization/?utm_source=chatgpt.com)  [github.com](https://github.com/qdrant/qdrant?utm_source=chatgpt.com) | Milvus IVF-PQ does similar compression but only inside the IVF index; if you later switch index types you must rebuild  [milvus.io](https://milvus.io/docs/ivf-pq.md?utm_source=chatgpt.com) |
| Simplicity for local dev & CI | One Docker command, no sidecars. Starts in <1 GB RAM. | Milvus needs etcd + Pulsar even for dev; Weaviate binary is light, but you don’t need its hybrid BM25 layer now. |
| Cloud-agnostic scale-out path | Since v0.8 Qdrant has distributed mode with Raft, sharding & replication; create more shards, add nodes, done  [qdrant.tech](https://qdrant.tech/documentation/guides/distributed_deployment/) | Milvus clustering is older/mature, but heavier to operate (stateful query/index nodes, segment leaders, etc.). |
| Feature extras you *don’t* need today | GPU search isn’t critical (you’ll GPU-accelerate *embedding* instead). | Milvus has GPU indexes (CAGRA, IVF-PQ)  [milvus.io](https://milvus.io/blog/introducing-the-milvus-sizing-tool-calculating-and-optimizing-your-milvus-deployment-resources.md?utm_source=chatgpt.com)  – handy only if search latency, not embedding, becomes the bottleneck. |
| Long-term exit ramp | If—in two years—you outgrow Qdrant’s cluster or want DiskANN-style SSD indexing, the schema/api concepts map cleanly to Milvus. |  |

### **Bottom line**

1. Local proof-of-concept: Qdrant’s single-binary Docker lets you load a few million Jina CLIP embeddings and hit the ground running in minutes.
2. Production launch: Stay on Qdrant, enable product quantization as data grows, and turn on distributed mode when one box isn’t enough. You still get rich payload filters and the same REST/gRPC API.
3. Future scale insurance: If you eventually need GPU-search or DiskANN, you can migrate the collection to Milvus; but you avoid its operational complexity until it genuinely pays off.

This keeps the stack simple, fast, and pragmatic—exactly what you asked for. We’ll proceed with Qdrant in the next step (install, load embeddings, run first searches).

Favicon

Favicon

Favicon

Sources

##### **You said:**

Ok. I'm a rust developer too so I may be biased when I said Qdrant. Are you sure it is the best choice?

###### **ChatGPT said:**

Thought for 13 seconds

Short answer: Yes—*for the stage you’re at and for the kind of queries you’ll run, Qdrant is the most practical first choice.*Below is the reasoning boiled down to the two things that really make or break performance for your workload.

## **1 Metadata-heavy, filter-first queries**

Every search you run will include narrow filters:

text

Copy

Edit

(user\_id = me OR is\_public = true) AND platform = "instagram"

With tens of millions of frames per platform those filters decide which few-million vectors the ANN layer should even touch.

| **What you need** | **How Qdrant does it** | **Why the others don’t match as well** |
| --- | --- | --- |
| Filters applied *inside* ANN traversal  (so you don’t waste time scoring vectors that will be rejected later) | “Payload-aware HNSW” prunes candidates as it walks the graph | Milvus and Weaviate fetch K neighbours then post-filter; when K must be large to hit recall, latency spikes |
| Boolean logic (AND / OR) for tenant + public blending | Native JSON payload + bool ops; single round-trip | Milvus needs two queries or id-join; Weaviate needs GraphQL union that’s slower on large sets |

For your *public-plus-per-user* access pattern this “filter-inside-ANN” detail is what keeps p95 latency in the double-digit-millisecond range rather than drifting into hundreds.

## **2 Memory footprint versus data volume**

Raw CLIP/JinaCLIP vectors → 768 dims × 4 bytes ≈ 3 KB each.  
1 billion frames ≈ 3 TB in RAM—no chance.

| What you need | Qdrant | Milvus | Weaviate |  
|---|---|---|  
| Drop RAM by >90 % without killing recall | Product or binary quantization (1 byte/val, optional re-rank on originals on SSD) | IVF-PQ (good), DiskANN (best) but forces you onto a heavier cluster stack | Early PQ plug-in; still RAM-heavy for 10⁹ size |  
| Stay dev-friendly | One Rust binary / Docker, same config in prod | Needs etcd + Pulsar, bigger ops surface | Light, but offers no SSD index yet |

Quantization is essential the moment you cross the hundred-million mark. Qdrant gives you that saving with zero external services and a tiny config flag, which is exactly the kind of “simple system” you asked for.

## **3 Future-proofing in case you really hit Google-scale**

| **Possible future need** | **Path if you start on Qdrant** |
| --- | --- |
| Horizontal scaling / HA | OSS distributed mode (Raft-backed) landed Q1 2025—same API, add nodes  [qdrant.tech](https://qdrant.tech/documentation/guides/distributed_deployment/?utm_source=chatgpt.com) |
| SSD-only billion-scale DiskANN | If latency/RAM still a problem, export IDs + vectors to Milvus’s DiskANN. Qdrant and Milvus share the same “vector + payload” mental model, so migration is a data copy, not an app rewrite. |
| Hybrid BM25+vector search | You can either front Qdrant with OpenSearch (keyword layer) or—if that becomes central—swap to Weaviate for those endpoints only. Nothing in your code ties you rigidly to one DB as long as you keep a thin repository interface. |

## **Bottom line**

*Qdrant keeps the first version of your system lightweight, Rust-native, and blisteringly fast on the filter-heavy queries you’ll run every minute of every day.* You won’t outgrow it until:

1. Data > a few billion vectors *and* you must fit everything on one cluster, *and/or*
2. You need ultra-low-latency (<10 ms) disk-backed search where DiskANN shines.

At that point you can still step over to Milvus with minimal re-work—but there is no operational pay-off in carrying Milvus’s extra moving parts from day 0.

So yes—stick with Qdrant, build the PoC, measure, and only switch if hard numbers tell you to.

END