Prompt:

Imagine you're the best dev ops on aws in the world. You work at Oriane. At Oriane we monitor videos with AI vision. Technically it means that we will have a monitored video, we will extract one frame for each second (e.g 60 frames for a 60 seconds video). Then we will do the same with a log of maybe 1 million videos download from instagram or tiktok. We will then compare all the frames of the monitored video with the frames of the watched videos in order to find matches. We use an AI model who's goal is to detect visual similarities. The problem that we have is that we want this analysis comparis millions (even billions of frames) to be performed quickly and at a low cost. We implemented with AWS SQS and lambda functions. Is there an even more scalable way?  
  
Response:  
  
Below is a conceptual playbook on how to scale beyond just using SQS + Lambda. The short version is that you want to move from “invoking one Lambda for each frame in SQS” to a more batch- or cluster-oriented approach, leveraging either a distributed compute framework (EMR, Spark, AWS Batch, HPC, or containers) and/or a vector similarity search architecture.

## **1. Stop doing N=1 frames in SQS+Lambda**

When you have to process millions or billions of frames, letting SQS trickle events one-by-one to Lambda is rarely the most optimal or cost-effective approach. Lambda can be fantastic for event-driven tasks, but it struggles under extremely heavy loads doing CPU- or GPU-bound tasks, especially if:

* You have to spin up a *lot* of parallel Lambdas (and hence pay for concurrency).
* You need a GPU. GPU-based Lambda support is not as straightforward.
* You want more control over the memory/CPU/gpu usage (Lambda is relatively limited).

### **The big overhead**

Each Lambda invocation has overhead in cold starts and container spin-up. Even with the new SnapStart or well-warmed Lambdas, this overhead can get magnified at scale if you're receiving thousands (or tens of thousands) of messages in a short window.

## **2. Move to a distributed (batch or streaming) approach**

### **Option A: AWS Batch (containers on ECS)**

AWS Batch is effectively a managed service that spins up ECS/Fargate (or EC2) container fleets for you. Some benefits:

1. **Batch scheduling**: You can dump your “list of frames to analyze” into a job queue, and AWS Batch handles queuing, scheduling, and scaling the container cluster to process them in parallel.
2. **Container-based**: You can bring your own Docker image with the AI model, meaning zero overhead on repeated “Lambda-like” cold starts. You pay only for the EC2 (or Fargate) underlying instances/time.
3. **More control over resources**: If you need GPU instances (e.g., P3 or G4DN), or you want massive parallel CPU tasks, you can do that easily with AWS Batch job definitions.

### **Option B: Spark on EMR or EKS**

If you want to do large-scale distributed computing, or if you need to do more advanced data engineering tasks around those frames, Spark on EMR is a natural choice. Key points:

1. **Distributed data**: You can store your frames on S3. EMR can read them in parallel.
2. **Massively parallel**: Spark can spin up clusters with hundreds or thousands of cores. You run a single job to transform all frames, generate embeddings, and store the results in a single pass.
3. **Cost management**: With Spot Instances, you can drastically reduce compute costs.
4. **GPU integration**: EMR can integrate GPU-based workloads as well, though it can get more complicated with specialized AI frameworks. But if you’re comfortable with Spark + MLlib or a PySpark-based approach, it can definitely scale.

### **Option C: HPC on AWS (AWS ParallelCluster)**

If you have a big HPC or supercomputing-like problem—like billions of frames to process in a short window with specialized libraries—AWS ParallelCluster can help you spin up HPC clusters with the right CPU/GPU configurations. This is typically used for large simulation or HPC tasks.

## **3. Vector search architecture (doing “approximate nearest neighbor” instead of a direct comparison)**

A big key to not going broke or slow: don’t literally compare all frames from the monitored video to all frames from the reference database, pixel by pixel or with a raw neural net feed. Instead, generate embeddings and then do an approximate nearest neighbor (ANN) search.

### **Step 1: Generate embeddings**

Your AI model presumably can output a vector (e.g., 512, 768, 1024 dimension) for each frame that captures its “visual signature.”

Use your distributed approach (EMR, ECS, AWS Batch, HPC, etc.) to create these embeddings once per frame in your large library (Instagram/TikTok frames). Store the embeddings somewhere:

* Amazon OpenSearch Service with vector indices
* Self-managed vector DB (e.g., Milvus, Faiss, Weaviate, Pinecone, etc.)
* Or even PostgreSQL with pgvector, though that might get unwieldy for billions of vectors

### **Step 2: For the “monitored video,” also generate embeddings per second (60 frames -> 60 vectors, for example).**

### **Step 3: Perform vector similarity search (ANN)**

Instead of “for each of the 60 frames, run a direct inference against billions,” you just do vector similarity lookups. A good vector DB or OpenSearch can handle real-time or near-real-time vector queries. The magic:

1. **ANN-based indexes** (HNSW, IVF, etc.) drastically cut down on the time complexity of searching billions of vectors.
2. You only pay for the compute needed to store the index and do the queries.
3. You can do “top-k” searches to retrieve the most visually similar frames quickly.

This architectural pivot is often the single biggest difference between a naive “run the model for every single frame pair” approach vs. a truly scalable approach. The net effect:

* You do a single offline embedding job for your entire reference database (billions of frames).
* You store that in a vector index.
* When you get a new monitored video, you embed it (just 60 frames for a 60-second video or however many you need).
* You query each of those embeddings against the vector index.
* The total time can be seconds/minutes, not days or hours.

## **4. Container Orchestration: ECS vs. EKS**

If the above approaches (AWS Batch or EMR) don’t fit your style, or you want more of a microservices / container approach:

* **Amazon ECS**: Easiest orchestration, simpler to set up, fully managed if you go “Fargate.” Good for short-ish and container-based tasks.
* **Amazon EKS**: Runs standard Kubernetes. More complicated but also more flexible if you want to integrate with a wide ecosystem of K8s tools.

Either way, you can schedule big container-based tasks that generate embeddings or do chunk-based matching. You’re not forced into the “one message -> one Lambda” pattern. You can do “thousands of frames -> single container job,” which is more efficient.

## **5. Orchestrating the workflow: Step Functions or Managed Airflow**

Whichever compute approach you pick (Batch, EMR, ECS/EKS, HPC), you’ll eventually want a robust way to orchestrate end-to-end:

1. Trigger the extraction of frames (maybe via an S3 event or a step in Step Functions).
2. Spin up your cluster / containers to embed those frames, store them in your vector database.
3. Then do the query or matching steps, etc.

**AWS Step Functions** is a common choice for serverless orchestrations, or you can use a more data engineering tool like **Managed Apache Airflow** if you’re used to the Airflow DAG model.

## **6. Summary of the recommended approach:**

1. **Generate embeddings for your massive reference library** (the billions of frames):  
   * Use ECS, AWS Batch, EMR, or HPC. Avoid a naive SQS+Lambda fan-out for each frame.
   * Store embeddings in a vector database that supports approximate nearest neighbor.
2. **When a new monitored video arrives**, extract frames (1 frame/second):  
   * Generate embeddings for those ~60 frames (again, in a container job or even a single quick Lambda if it’s not too big).
   * Issue ~60 vector similarity searches to the vector DB.
3. **Return your top matches** efficiently, skipping naive brute force.

This architecture—**embedding + vector search**—is by far the most scalable approach to quickly find duplicates or near-duplicates among billions of images.

### **Practical Examples**

* **Amazon OpenSearch**: If you want an AWS-native vector store, recent versions let you store vectors and do kNN or approximate kNN. Perfect for large-scale search.
* **Self-managed**: Tools like **Milvus** or **Faiss** on ECS/EKS.
* **Managed service**: Tools like Pinecone (though that’s 3rd-party, not purely AWS) also can handle large-scale vector search.

## **Final Takeaways**

1. **Yes, there is a more scalable way** than SQS + Lambda for billions of frames.
2. **Consider HPC or a distributed batch approach** (AWS Batch, ECS, EMR, etc.).
3. **Use embedding + vector search** to drastically reduce your computational load when searching.
4. **Orchestrate** with Step Functions or Managed Airflow so you have end-to-end control and reliability.

Implementing a flow like this will significantly reduce cost and latency, because you’re harnessing the power of distributed compute *and* you’re replacing brute-force frame-to-frame matching with modern approximate nearest neighbor indexing.