TwelveLabs

WHITEPAPER

A Guidance on Multi-Vector Video Search with TwelveLabs Marengo

Eric Kim, Staff Solutions Architect



WHITEPAPER

TABLE OF CONTENTS

1. INTRODUCTION

1.1 - Why Marengo Outputs Multiple Vectors
1.2 The Multi-Vector Challenge

2. APPROACH 1: FUSED EMBEDDINGS

2.1 - Overview
2.2 - Technical Implementation
2.3 - Advantages
2.4 - Drawbacks
2.5 - Recommendation

3. APPROACH 2: MULTI-VECTOR RETRIEVAL

3.1 - Overview
3.2 - Technical Implementation
3.3 - Advantages
3.4 - Drawbacks
3.5 - Recommendation

4. DEEP DIVE INTO DETERMINING MODALITY WEIGHTS FOR MULTI-VECTOR QUERY ROUTING

4.1 - Fixed Weights with Statistical Optimization
4.2 - Dynamic Weights with Specialized ML Model
4.3 - Intent-based Dynamic Query Routing

5. CONCLUSION

5.1 - Comparative Summary of Approaches
5.2 - Best Practices
5.3 - Closing Thoughts

6. ABOUT TWELVE

1.1 - Why Marengo Outputs Multiple Vectors

Marengo is TwelveLabs' multimodal embedding foundation model for **production-grade video intelligence.** It powers our Embed API and Search API, and it is explicitly designed for real-world retrieval constraints—**multilingual archives, composed (image + text) queries, long-form video**, and **specialized domains like sports**—not just short-video, English-only benchmarks.

From first principles: **video is not one signal**. It is a synchronized bundle of (1) what happens visually over time, (2) what is heard (including non-speech sound), and (3) what is said (speech content). Collapsing these into a single undifferentiated vector forces a model to "average" incompatible evidence types, which is exactly how retrieval systems lose precision: dialogue queries get diluted by visuals, and sound-event queries get drowned out by scenes.

Marengo 3.0 makes this separation explicit at the embedding interface. When you create video embeddings, TwelveLabs can return three embedding types:

- Visual: based on visual content
- Audio: based on audio content (sounds, music, silence, mood, tone)
- Transcription: based on spoken words extracted from the audio track (dialogue, speech, voiceover, narration)

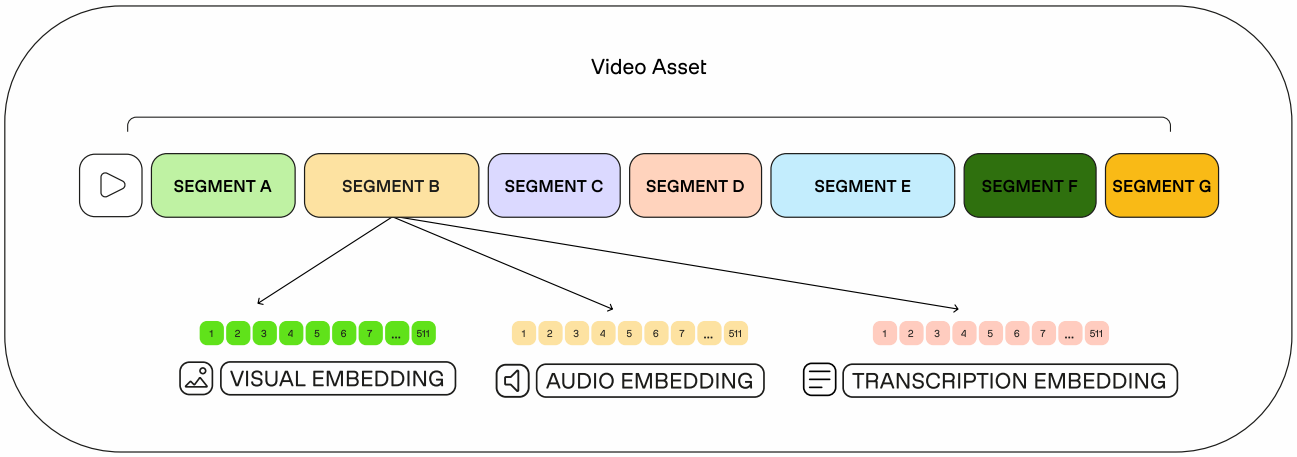


Figure 1. Visual diagram representing the decomposition of a video asset into segments, and the multi-vector representation of each segment output by Marengo 3.0.

This is a deliberate API design choice: it gives our developer users **separable, inspectable "channels"** that map cleanly to user intent. "Show me the moment the crowd erupts" should primarily hit **audio**; "find the clip where they say 'net revenue retention'" should primarily hit **transcription**; "find the play where the defender slides and blocks the shot" should primarily hit **visual**.

While the **audio** embedding encapsulates speech elements in the audio track, including tone and intonation; the **transcription** embedding provides a better optimized performance for searching for an exact spoken line, similar phrases, or a topic of discussion. This distinction means the transcription weight is the knob that most directly affects "who said what" retrieval performance.

What changed with Marengo 3.0 (and why it matters for enterprise retrieval)
Compared to previous model versions, Marengo 3.0 also modernizes the practical economics of multi-vector systems:

- **Optimized 512-dimension embeddings** (smaller vectors reduce storage and accelerate search)
- **New transcription embedding** for semantic search across speech and dialogue present in the audio.
- **Long content support up to 4 hours** (critical for enterprise libraries: games, trainings, meetings)
- **Extended text processing up to 500 tokens** for queries and text embeddings (enables more descriptive, complex queries without truncation)
- **Expanded language support** (36 languages + English) to make cross-region archives searchable without rewriting queries per language

Finally, while Marengo returns modality-specific embeddings, TwelveLabs generates embeddings in a **shared latent space** across modalities, enabling "any-to-any" retrieval patterns (e.g., text → video, image → video), which is foundational for multimodal search experiences.

1.2 The Multi-Vector Challenge

Multi-vector embeddings are powerful because they preserve modality-specific signal. They are challenging because they force you to design a system that answers one deceptively hard question:

When a user asks something in natural language (or image + text), which modality should dominate the retrieval—and how do you combine evidence across modalities without losing precision?

With Marengo 3.0, this challenge becomes more important (not less), because the model enables more retrieval behaviors in production:

- **Composed multimodal queries** (image + text) allow users to disambiguate intent. That means your query can carry both visual identity and semantic intent simultaneously.
  "this player" + "blocking a goal"
- **Long-form video** (up to 4 hours) means you will generate and store many more clip-level embeddings, increasing both index size and the need for robust ranking across segments.
- **Longer queries** (500 tokens) mean enterprise users can express constraints that span modalities ("Find the clip where the presenter says X while the slide shows Y and the audience laughs"), which is exactly where naive single-vector search tends to break down.

1.2.1 What makes multi-vector search hard?

When you embed a video, you don't get "one vector per video." You can choose:

- **Embedding scope:** clip-level (per segment) vs asset-level (average of all segments)
- **Segmentation strategy:** dynamic (scene-based) vs fixed (time-based)
- **Embedding options:** visual, audio, transcription (or any combination of the three)

So the system design problem is not just "vector search." It is **multi-index retrieval + temporal grounding + evidence fusion:**

1. Indexing & storage

You now have multiple embeddings per clip (and potentially many clips per asset). You must decide whether to store these embeddings in one fused index or separate modality indices, and how to map results back to (video_id, start_sec, end_sec) for precise timestamps.

2. Query interpretation

A query is rarely "purely visual" or "purely transcription." Even simple enterprise queries often combine constraints:

- "Find all videos of a specific person giving a talk" (visual entity + action)
- "Find the clip where she says 'Q3 pipeline'" (transcription)
- "Find the moment the alarm goes off" (audio)
- TwelveLabs + "Find this logo appearing in other clips, even in different settings (backgrounds, lighting, camera angles)" (visual, fine-grained)

Without routing, you either (a) search the wrong modality or (b) search all modalities and get noisy ranking.

3. Ranking & calibration

When you do search across multiple modalities, you must decide how to combine similarity signals into one ranked list. This is where most multimodal retrieval systems fail in practice: they either hard-code one modality (usually visuals or text) or average everything, which penalizes the modality that actually carries the answer.

01 Introduction

1.2.2 Best Practices for Multi-Vector Search with Marengo

Multi-vector architectures historically presented challenges due to the complexity of managing multiple vector indices and merging the outputs of multiple vector types.

This whitepaper's goal is to turn that capability into concrete system guidance. We cover two foundational architectures—**fused embeddings** and **multi-vector retrieval** and then dive deep into strategies for determining modality weights in multi-vector systems. By the end, you will have the framework to choose the right approach for your query distribution, team capabilities, and operational constraints—so you can build video search that is robust across visual, audio, and spoken-language intent, at enterprise scale.

02 Approach 1: Fused Embeddings

2.1 - Overview

The simplest way to operationalize Marengo's multimodal embeddings is to **fuse the multi-vector output into a single embedding per video segment**, then index that single embedding in your vector database. Marengo 3.0 returns three 512-dimension embeddings per segment (one each for visual, audio, and transcription)—a simplified output format that makes fusion straightforward and consistent across deployments.

2.2 - Technical Implementation

In the fused approach, we compute a single vector representation E_{fused} from the three modality embeddings:

$$E_{fused} = \frac{w_v \cdot E_{visual} + w_a \cdot E_{audio} + w_t \cdot E_{transcript}}{\left\| w_v \cdot E_{visual} + w_a \cdot E_{audio} + w_t \cdot E_{transcript} \right\|_2}$$

Equation 1. Fused embedding formula

For visually-heavy video retrieval benchmarks and many visual-centric archives, TwelveLabs' recommended baseline fusion weights are:

- $w_v = 0.8$ (visual)
- $w_a = 0.1$ (audio)
- $w_t = 0.05$ (transcription)

Then we L2-normalize the fused vector to unit norm before indexing. This results in a single 512-dimension fused embedding per video segment.

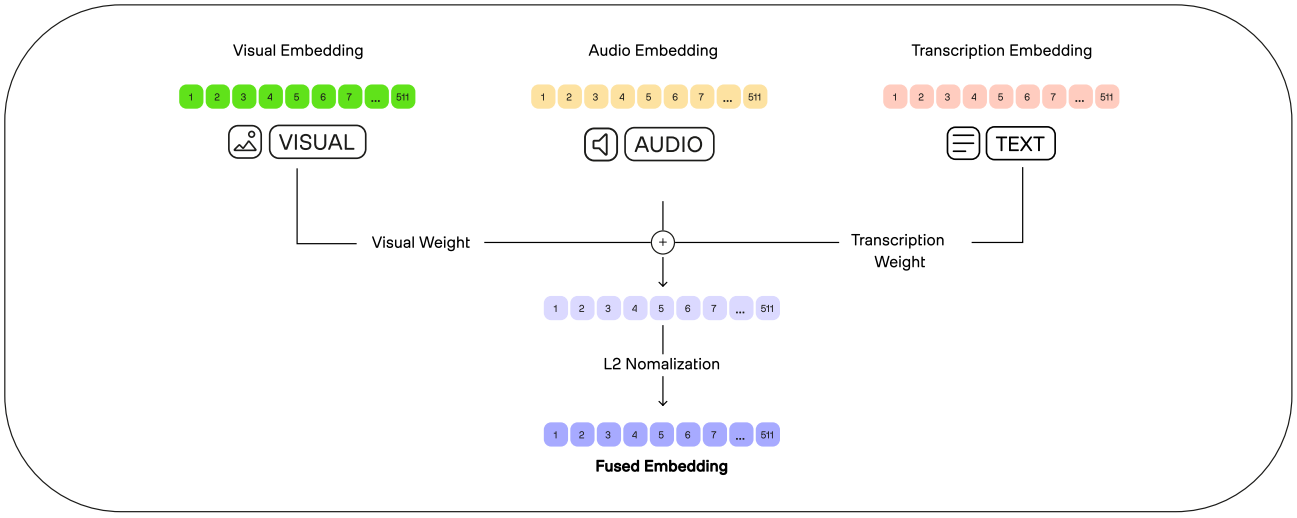


Figure 2. Diagram representing the multi-vector fusion process.

At a high level, the fused pipeline looks like:

1. Segment the video into searchable units (clips/segments).

- Marengo by default dynamically segments video assets into segments based on shot boundaries. The segmentation strategy can be separately configured depending on the use case.

2. Extract embeddings per segment with Marengo.

- Retrieve the three vectors $E_{visual}, E_{audio}, E_{transcript}$, each 512-d in the case of Marengo 3.0. If a vector modality doesn't exist for a given input, set the respective vector to 0.

3. Fuse + normalize once during ingestion.

- Compute E_{fused} using the fused embedding formula (Equation 1).
- Index only E_{fused} as the segment representation in your vector database.

4. Search by embedding the user query and retrieving nearest neighbors of E_{fused}.

- For text queries, embed the text and retrieve via cosine similarity (or dot product if all vectors are normalized).
- For composed multimodal queries, embed the image+text query into a single query embedding and search the same fused index.

2.3 - Advantages

Fused embeddings are still the fastest path to a working, sufficient baseline:

- **Single index, single retrieval pass**. One vector per segment means no multi-index orchestration and no score-fusion logic at query time.
- **Works similarly to single-modality embedding models**. Designing and implementing a single, fused embedding architecture follows the same patterns as existing text-based retrieval augmented generation (RAG) systems.

- **Operational simplicity for long-form content**. If you're indexing long videos (hours), your segment count rises quickly. Fusion helps keep the vector DB footprint and query fan-out minimal, which can matter early in a deployment.

2.4 - Drawbacks

Fusion intentionally introduces a **bias**: you are deciding—once—what the system should "care about."

- **Fixed weights do not adapt to query intent**. This is a fundamental limitation. A system tuned for visually-heavy retrieval will systematically underperform for speech-heavy queries ("find where she says X") or audio-event queries ("find the moment the alarm goes off"). Although the fixed weights can be statistically optimized for historical queries, it will not be able to adapt on a query-by-query basis nor to the changing user query behavior over time.
- **Fusion can obscure failure modes**. When retrieval quality drops, you can't easily isolate whether the issue comes from (a) the weight choice, (b) modality-specific embedding quality for your domain, (c) segmentation decisions, or (d) the fusion operation itself.
- **Fused embeddings are irreversible**. The process of fusing multiple embeddings into a single embedding is a one-way process. Upgrading to a multi-vector architecture (to be discussed in subsequent sections) will require either reprocessing the video assets to obtain the original 3 separate embedding types or storing the original 3 separate embedding types, which duplicates storage costs.

2.5 - Recommendation

Best for

- Teams with prior experience building single-vector RAG architectures who want the simplest, **lowest-risk architecture** that still delivers sufficient semantic retrieval quality.
- Deployments where queries are predominantly single-modality and the added benefit of multi-vector search is minimal.
- Early-stage or benchmarking workflows where you want a familiar baseline before introducing more moving parts.

Not recommended for

- Teams aiming to build true state-of-the-art (SOTA) video semantic search.
- Use cases where query intent is widely varied across the different modalities.
- Enterprise deployments where debugging, auditability, and modality-specific tuning are requirements.

Generally, **fusion is the "single-index baseline"**—mostly sufficient when your query distribution is stable and focused on a single modality.

3.1 - Overview

This approach keeps Marengo's multimodal embeddings **separate** and combines them only at search time. Instead of fusing vectors (Approach 1, Section 2), you build **three vector indices**—one per embedding type—and apply weights to combine similarity scores:

- **Visual index** (what happens on screen)
- **Audio index** (sounds in the audio like music, sirens, applause, mood, tone)
- **Transcription index** (spoken words extracted from the audio track)

Conceptually, separation gives you control. You can see (and debug) whether a result matched because of visuals, audio, or dialogue—without introducing the hard-to-reason-about geometry effects of vector fusion.

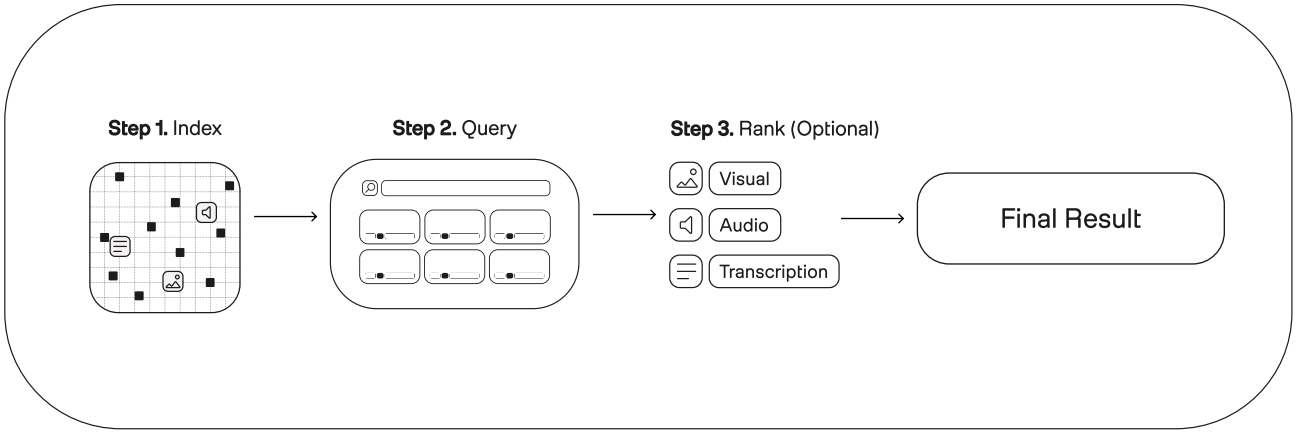


Figure 3. Diagram representing the key steps to multi-vector retrieval.

At a high-level, there are 3 key steps to the multi-vector retrieval process:
1. Index - Selecting the strategy for storing multiple modality embeddings and determining the optimal database schema based on anticipated access patterns (Section 3.2.1).
2. Query - Structuring the database queries based on the database schema from the previous indexing step to retrieve results from respective embedding modalities (Section 3.2.2).
3. (**Optional) Rank** - Depending on the query strategy, the results from multiple database queries need to be merged and sorted to output a final ranking of the results (Section 3.2.3).

3.2 - Technical Implementation

3.2.1 - Indexing

For each video, you still choose a segmentation strategy and embedding scope, but now you persist each modality embedding $E_{visual}, E_{audio}, E_{transcript}$ The exact database schemas and design can become more complex depending on requirements for production deployments, but they will largely fall into 2 patterns.

1. Single index with distinguished modalities - Store all modalities into a single vector index, but specify the embedding type in the database query during retrieval.

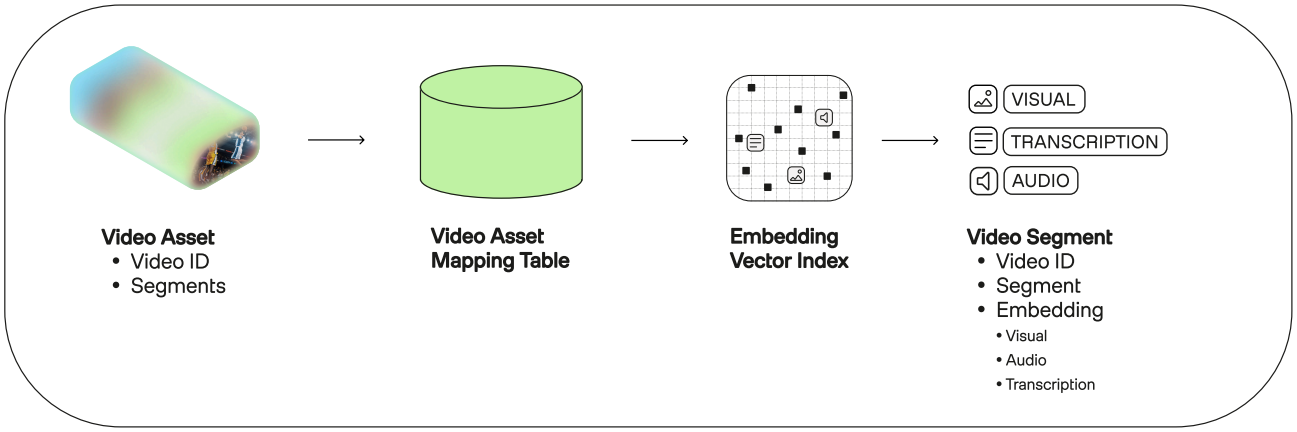


Figure 4. Diagram representing an example database setup with all embedding modalities stored in a single vector index.

Database schemas can vary depending on the use case and requirements. Figure 4 illustrates a minimal configuration for storing all embedding modalities in a single vector index. Although the 3 embedding modalities are stored in the same vector index, it's important to distinguish between the different modalities for querying separately.

2. Separate indices - Each modality as a separate vector index (visual, audio, transcription).

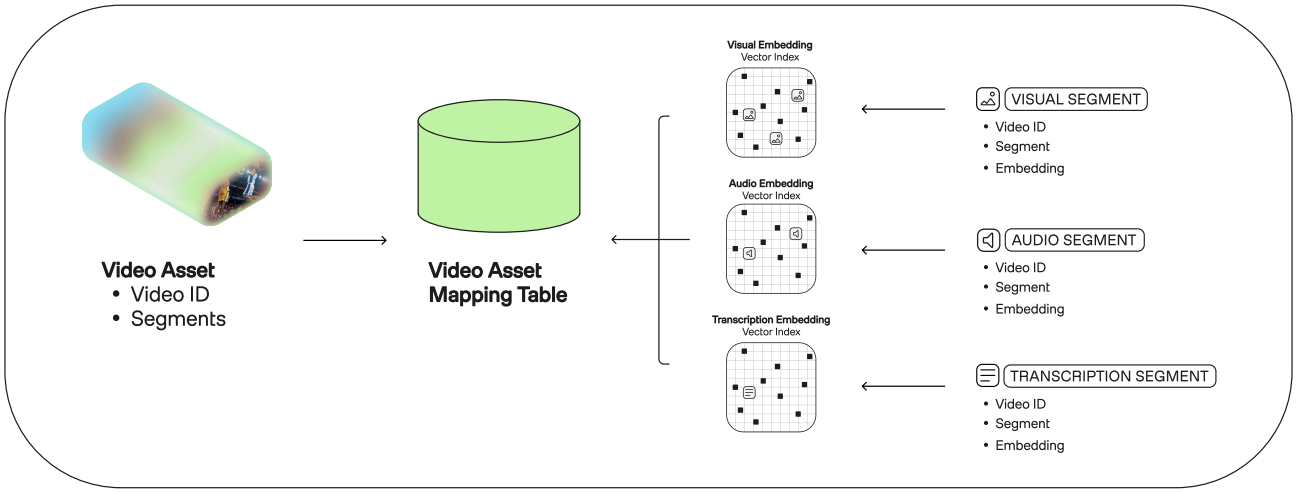


Figure 5. Diagram representing an example database setup with separate vector indices per embedding modality.

Unlike the single index database setup, Figure 5 depicts the minimal configuration for storing the 3 embedding modalities in respective vector indices. Separate vector indices may be required for vector database engines that don't support storing and querying multiple vectors types in a single index. This setup can also be advantageous for scaling the underlying databases for each embedding type independently, depending on the usage patterns.

3.2.2 - Querying

At query time, run **separate, parallel searches**, then combine candidates into a single result. If a user already knows which modalities to leverage, manual configurations can be used to select any combination of the visual, audio, and transcription indices.

There are 3 strategies for querying multiple vector indices. These strategies can be implemented in combination to enable configurability for end users.

1. Direct selection of a single modality.
Enable users to explicitly select a single vector modality with their query. This approach only works if the user knows the corpus of videos they are searching across or has a focused query in a single modality.

Take these scenarios for example:

- When searching for queries that are clearly a single modality like "a red car on the side of the road", the user can optionally select the visual embedding type as the search scope.
- If a user uploads a single image to perform an image-to-video search, the system can select only the visual embedding type as the search scope on behalf of the user.

Whether it's through direct user selection or indirect system selection, choosing a single modality will enable skipping the ranking step (Section 3.2.3) altogether.

2. One query embedding across multiple modalities.

Marengo embeddings are designed to live in a **shared latent space across modalities**, enabling "any-to-any" retrieval. Practically, that means you can embed the user's text query once, then use that same query embedding to search a combination of the visual, audio, and transcription indices.

For example, a query like "an explosion" can be relevant in all 3 vector modalities:

- **Visual** - a sudden, bright flash followed by a fireball with destruction of physical objects in the visual frames of the video
- **Audio** - a sudden, loud sound resembling that of an explosion in the audio
- **Transcription** - a person describing the Big Bang theory and using the word "explosion" in the explanation.

Using the same query embedding across the 3 modalities will enable retrieval of all 3 types of relevant videos.

3. Query decomposition using LLMs.

Similar to how lexical, tag-based search systems can use operators like AND and OR to create complex filter-based queries, natural language queries can also be composed of multiple components.

Take this complex semantic query as an example:
"A basketball player dunking in an empty stadium with beats playing in the background while a narrator discusses the importance of high school basketball programs"

This query can be used as a single query embedding like above, or it can be separated out into 3 distinct queries using a large language model (LLM):

- Visual query: "A basketball player dunking in a empty stadium"
- Audio query: "beats playing in the background while a narrator discusses"
- Transcription query: "the importance of high school basketball programs"

After the decomposition, the separately embed decomposed queries and use the embeddings to query the respective vector modalities.

Important Caveat: do not ask the LLM to output numeric modality weights
LLMs can explain intent well, but free-form numeric weights are not grounded or reliably calibrated. Even small changes in prompt wording can yield different weight "guesses," which makes system behavior non-repeatable and hard to debug. The LLM's role should be: **decompose and translate**, not invent weights.

3.2.3 - Ranking

Querying each vector modality in parallel introduces the challenge of merging and ranking the parallel results.

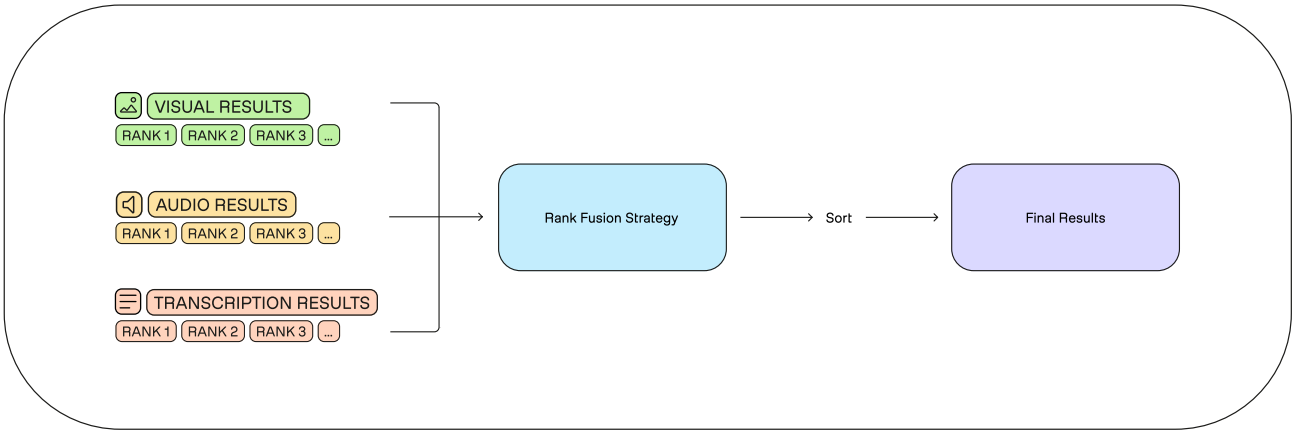


Figure 6. Diagram representing the process of taking the query results of the 3 embedding modalities and applying a rank fusion strategy.

A rank fusion strategy determines the way in which the parallel modality query results come together into a single ranking. Based on a formula, score and sort the returned video segments into a single list. Building a rank fusion strategy and optimizing it can become complex. These 2 strategies serve as the foundation for building the right one for your use case:

1. Rank-based fusion

Not to be confused with the fused embeddings approach, rank-based fusion as a strategy relies only on the rank positions of video segments in their respective vector modality query results to create a single, merged result. A common rank fusion formula is Reciprocal Rank Fusion (RRF).

$$\text{score}_{RRF}(s) = \sum m \in M \frac{1}{k + \text{rank}_m(s)}$$

Equation 2. Reciprocal Rank Fusion (RRF) formula.

Where:

- s represents the video segment
- m represents a vector modality in $M = [visual,\ audio,\ transcript]$
- $rank_m(s)$ is the rank of video segment s in the ranked list for modality m. If s is not in a list, set to ∞
- k is a non-negative constant smoothing parameter. Commonly set to k = 60, but optimal values can vary depending on the use case.

2. Score-based fusion

The formula for merging parallel vector query results using similarity scores looks like:

$$score(s) = \sum_{m \in M} w_x \cdot sim(Q_m, E_m(s))$$

Equation 3. Formula for merging the parallel vector searches across visual, audio, and transcription embedding types.

Where:

- s represents the video segment
- m represents a vector modality in *M* = [*visual*, *audio*, *transcript*]
- w_m represents the weight for the vector modality
- Q_m is the query embedding for modality m
- $E_m(s)$ is the m modality embedding of video segment
- $sim(A, B)$ represents the cosine similarity between 2 vectors A and B

In comparison to the rank-based fusion approach, combining scores from parallel searches requires more involved optimization, an understanding of the scores in each modality, and how they compare against one another. Specific strategies for determining the modality weights will be discussed in the Section 4.

3.3 - Advantages

Multi-vector retrieval unlocks several fundamental benefits that stem from preserving modality separation throughout the search pipeline:

- **Preserves modality-specific signal fidelity**. Each embedding type (visual, audio, transcription) retains its full representational power in its own index. Unlike fusion, where distinct signals are collapsed into a single geometric space, multi-vector retrieval avoids the information loss and "averaging" artifacts that occur when semantically different modalities compete within one vector.
- **Transparent, modality-level debuggability**. When retrieval quality degrades or a result is unexpectedly ranked, you can isolate the cause: "Did the visual match carry this result? Was it the transcript similarity? Did non-speech audio contribute?" This level of inspection is invaluable for enterprise deployments where explaining why a result appeared matters as much as the result itself.
- **Decouples indexing decisions from ranking decisions**. With fused embeddings, the weighting scheme is baked in at ingestion time—changing it means reprocessing your entire corpus. Multi-vector retrieval moves that decision to query time, which means you can adjust modality weights, experiment with fusion strategies, or respond to shifts in query behavior without re-indexing a single video.
- **Supports modality-specific optimization**. Because each index is independent, you can tune or scale them separately. If your workload is transcription-heavy, you can allocate more resources to that index. If your audio corpus is sparse, you can configure that index differently. This flexibility is difficult (or impossible) with a single fused representation.
- **Foundation for adaptive architectures**. Multi-vector retrieval is the prerequisite for query-adaptive systems. Once your indices are separate, upgrading to dynamic routing, intent-based weighting, or LLM-driven query decomposition becomes a query-time problem—not a data migration. The indexing work you do today carries forward as your retrieval sophistication grows.

3.4 - Drawbacks

Multi-vector retrieval trades the simplicity of a single index for greater control—but that control comes with real operational and design costs.

- **Increased infrastructure complexity**. Three indices means three times the write paths, three times the index management, and three times the failure surface. For teams accustomed to single-index RAG architectures, this is a meaningful step up in operational burden—more shards to monitor, more replication to configure, and more capacity planning to get right.
- **Higher storage footprint**. Storing separate embeddings for visual, audio, and transcription per segment multiplies your vector storage requirements compared to a single fused embedding. For large video libraries with many segments per asset, this cost accumulates quickly and must be factored into infrastructure budgets.

- **Query-time latency and fan-out**. Retrieving from multiple indices—whether sequentially or in parallel—adds latency compared to a single-index lookup. Even with parallel execution, you must wait for the slowest modality index to return before merging results. At scale, this fan-out pattern requires careful tuning to meet latency SLAs.
- **Score calibration is non-trivial**. Similarity scores from different modality indices are not inherently comparable. A cosine similarity of 0.85 from the visual index does not "mean the same thing" as 0.85 from the transcription index. Naively combining raw scores can produce unintuitive rankings, and building calibration logic (normalization, learned adjustments, or rank-based fusion) adds another layer of design complexity.
- **Requires explicit fusion/ranking logic**. Unlike fused embeddings—where the ranking is implicit in the single similarity score—multi-vector retrieval forces you to decide how to combine evidence from each modality. This is a design surface that does not exist in single-index systems, and getting it wrong can degrade relevance in ways that are harder to diagnose than a simple "bad embedding."
- **Evaluation becomes multi-dimensional.** Measuring retrieval quality now requires understanding each modality's contribution. A relevance drop could stem from the visual index, the transcription index, the fusion strategy, or the interaction between them. Building an evaluation harness that can isolate these factors is more work than evaluating a single retrieval path.

3.5 - Recommendation

Best for

- Teams that require **transparency and debuggability** in their retrieval pipeline—where understanding why a result matched (visual vs. audio vs. transcription) is a product or compliance requirement.
- Deployments with **mixed query intent** across modalities, where a single fused representation would systematically underweight the signals that matter for a meaningful portion of queries.
- Organizations building toward **state-of-the-art video semantic search,** where the ability to tune, calibrate, and adapt modality weighting over time is essential to long-term retrieval quality.
- Enterprise environments where **modality-specific tuning and scaling** may be required—for example, scaling the transcription index independently for dialogue-heavy workloads.

Not recommended for

- Teams seeking the **fastest path to production** with minimal infrastructure overhead. If your priority is simplicity and your queries are predominantly single-modality, fused embeddings (Approach 1, Section 2) will get you there with less operational burden.
- Deployments where **storage and latency budgets are tightly constrained** and the added cost of maintaining three indices cannot be justified by the retrieval quality gains.
- Early-stage projects where **query distributions are unknown** and you lack the evaluation infrastructure to measure whether multi-vector retrieval is actually improving relevance over a simpler baseline.

03 Approach 2: Multi-Vector Retrieval

Generally, **multi-vector retrieval is the production-grade foundation** for teams that need more than a single fused embedding can offer. Once the multi-index architecture is in place, the choice of weighting strategy—fixed weights, statistically optimized weights, or fully dynamic query routing—becomes a tuning decision rather than a re-architecture. Section 4 explores these weighting strategies in depth.

04 Deep Dive into Determining Modality Weights for Multi-Vector Query Routing

4.1 - Fixed Weights with Statistical Optimization

4.1.1 - High-Level Approach

Statistical optimization treats modality weight selection as a search problem: given a corpus of videos, a set of representative queries, and ground truth relevance labels, find the weight combination (w_v, w_a, w_t) that maximizes retrieval quality.

The process follows a straightforward workflow:

1. Assemble an evaluation dataset.
Collect a set of queries that represent the expected usage patterns for your application. For each query, label the ground truth results—the video segments that should be returned as relevant. The quality of this evaluation set directly determines the quality of your optimized weights. A small, carefully curated set (50–100 queries with reliable labels) is more valuable than a large, noisy one.

2. Define a retrieval quality metric.
Choose the metric you want to optimize. Common choices include precision (how many returned results are relevant), recall (how many relevant results are returned), or combined measures like F1 score or weighed metrics like mean average precision (mAP). The right metric depends on your application: if users care most about the top result, optimize for precision@1; if comprehensive retrieval matters, optimize for recall@K.

3. Run the optimization.
Systematically evaluate different weight combinations against your evaluation set. For each candidate (w_v, w_a, w_t) , execute the multi-vector retrieval pipeline, compute the quality metric, and record the result. The search can be as simple as a grid search over a discretized weight space (e.g., weights in increments of 0.1) or as sophisticated as Bayesian optimization for faster convergence. The goal is to find the weight combination that yields the highest metric score across your query set.

4. Validate on held-out queries.
Before deploying the optimized weights, validate them against a separate set of queries that were not used during optimization. This guards against overfitting—weights that perform well on the optimization set but fail to generalize to real-world query diversity. If performance degrades significantly on the held-out set, revisit your evaluation dataset or consider whether fixed weights are sufficient for your query distribution.

5. Deploy and monitor.
Once validated, deploy the optimized weights as your production configuration. Importantly, statistical optimization produces a point-in-time snapshot of optimal weights. As your content library grows, your user base shifts, or query patterns evolve, the optimal weights may drift. Periodic re-evaluation—using updated query samples and relevance labels—ensures your weights remain well-calibrated over time.

Based on TwelveLabs' evaluation on representative user queries, the same weights from Section 2 can be used as a baseline for a visual-heavy query distribution:

- $w_v = 0.8$ (visual)
- $w_a = 0.1$ (audio)
- $w_t = 0.05$ (visual)

4.1.2 - Advantages

- **Avoids fusion artifacts**. Each modality remains in its own index, so you don't create a single vector whose meaning is hard to interpret or debug.
- **Better debuggability**. When a result is wrong, you can answer: "Was it visually similar? Did non-speech audio drive this? Did the transcript match?"
- **A better stepping stone to dynamic routing**. Once you already maintain separate indices (unlike the fused embedding approach in Section 2), upgrading to dynamic weights becomes a query-time decision problem—not a data re-indexing problem.
- **Accessible to teams with foundational data science skills**. Unlike training a specialized ML router, statistical optimization relies on familiar concepts—evaluation metrics, grid search, cross-validation—that most data teams already understand. The workflow does not require deep ML expertise or specialized infrastructure, making it a practical option for organizations that have data science capability but are not staffed to build and maintain custom models.

4.1.3 - Drawbacks

- **More infrastructure moving parts**. Three indices means more writes, more storage, and more query fan-out than a single fused index.
- **Requires data science capability to execute effectively**. While the process does not demand deep ML expertise, it does require a team that can assemble representative evaluation datasets, define appropriate quality metrics, run systematic optimization, and validate results against held-out queries. Organizations without this capability—or without the bandwidth to dedicate to the effort—may find it difficult to realize the benefits of statistical optimization over simpler default weights.
- **Still not query-adaptive**. The weights are fixed. A query like "what did she say about Q4 pipeline" should heavily emphasize transcription, but static visually-skewed weights will still bias toward visuals unless you tune weights for your query distribution.

4.1.4 - Recommendations

This approach works well when your query distribution is **stable and reasonably homogeneous**. If most queries favor similar modality balances, a single optimized weight set can capture that pattern effectively. However, if your query distribution is highly diverse—spanning visual-heavy, dialogue-heavy, and audio-event queries in significant proportions—no single weight combination will satisfy all cases, and dynamic routing (Sections 4.2 and 4.3) becomes the more appropriate strategy.

Best for

- Customers who want **higher clarity and debuggability** than fused embeddings, but don't want the orchestration complexity of dynamic routing yet.
- Teams that expect most queries to remain **modality-skewed** (often visual) but need the ability to validate and tune modality contributions explicitly.
- Deployments where you want to **make the "indexing decision" only once**. This sets up the foundations to build up to a dynamic query routing approach in the future.
- Organizations with **foundational data science capability** that can dedicate time to assembling evaluation datasets, running optimization workflows, and validating results—but that lack the ML infrastructure or expertise to train and maintain a specialized routing model.

Not ideal for

- Customers with no **expectation of query behaviors or empirical data** for performing statistical optimization.
- Use cases with **highly variable query intent** across combinations of the 3 embedding modalities, where static weights won't satisfy the majority of input queries.
- Workloads where the **query distribution is unknown** and you need an adaptive system at query-time.
- Teams **without data science resources** to execute the optimization process. If you lack the bandwidth or expertise to build evaluation sets, define metrics, and run systematic experiments, simpler default weights or a productized routing solution may be more practical.

4.2 - Dynamic Weights with Specialized ML Model

4.2.1 - High-Level Approach

A traditional approach is to train a small model that maps either the query text (or Marengo embedding of the query) → modality weights (or a modality label). A specialized ML router treats modality weight prediction as a **supervised learning problem:** given a query, predict the optimal modality weights (or a discrete modality label) that will maximize retrieval quality for that specific query.

Unlike statistical optimization (Section 4.1), which finds a single weight combination for all queries, a learned router produces **per-query weights**—enabling the system to adapt its retrieval behavior based on the characteristics of each incoming query.

The process follows a typical ML workflow:

1. Collect and label training data.
Assemble a dataset of queries paired with their optimal modality weights or modality labels. This is the most labor-intensive step. For each query, you need to determine which modality configuration produces the best retrieval results—either through manual labeling, user feedback signals, or automated evaluation against ground truth. The diversity and quality of this training set determines how well the router generalizes to real-world queries.

2. Define the model architecture.
Choose a model that maps query representations to routing decisions. Common approaches include lightweight classifiers (predicting a discrete modality label) or regression models (predicting continuous weights). The input is typically the query embedding or a learned representation of the query text. The model should be small enough to run at query time without introducing significant latency.

3. Train and validate the model.
Train the router on your labeled dataset using standard supervised learning techniques. Split your data into training, validation, and test sets to monitor for overfitting. Evaluate the router not just on classification or regression accuracy, but on end-to-end retrieval quality—does the router's predicted weighting actually improve precision and recall compared to fixed weights?

4. Deploy and integrate into the retrieval pipeline.
Once trained, the router sits in front of your multi-vector retrieval system. At query time, the model predicts modality weights, which are then used to execute the weighted search across indices. This adds a model inference step to every query, so latency and reliability become operational considerations.

5. Monitor, retrain, and maintain.
Learned routers inherit the biases present in their training data. As your content library evolves, your user base shifts, or new query patterns emerge, the router's predictions may drift out of alignment with actual user intent. This requires ongoing monitoring of retrieval quality and periodic retraining on fresh data—an operational commitment that persists for the lifetime of the system.

This approach can deliver strong query-adaptive performance when the training data accurately represents production query patterns and when the team has the ML expertise to build, validate, and maintain the model over time. However, the upfront investment and ongoing maintenance burden make it a significant commitment compared to non-learned alternatives.

4.2.2 - Advantages

- **True query-adaptive behavior**. Unlike fixed weights, a learned router produces different modality weightings for different queries. A query about spoken dialogue routes toward transcription; a query about visual action routes toward visual. This per-query adaptivity is the core promise of dynamic routing, and a well-trained model can capture nuanced patterns that rule-based or anchor-based approaches may miss.
- **Single inference component in deployment**. Once trained, the router is a self-contained model that takes a query and outputs weights. There is no need for external LLM calls, no prompt engineering to maintain, and no anchor configurations to tune. This simplicity can be attractive for teams that prefer a "train once, deploy, and run" operational model over systems that require ongoing configuration management.
- **Can learn domain-specific patterns**. A router trained on your actual query distribution can capture routing behaviors that are specific to your content and users—patterns that may be difficult to articulate as explicit rules or anchors. If your users consistently phrase visual queries in a particular way, or if certain keywords reliably indicate transcription intent, the model can learn these signals directly from data.
- **Potential for high precision at scale**. With sufficient training data and careful model design, a learned router can achieve routing accuracy that exceeds heuristic or anchor-based methods—particularly for query distributions that are complex but consistent. For large-scale deployments with stable query patterns, the upfront investment in training can pay off in sustained retrieval quality.

4.2.3 - Drawbacks

- **Requires dedicated ML expertise**. Building a learned router is not a configuration task—it is a machine learning project. Your team needs experience with model architecture selection, training pipelines, hyperparameter tuning, and evaluation methodology. Organizations without this expertise will need to hire for it or invest in upskilling, which adds cost and timeline before any routing improvements materialize.

- **Significant data labeling burden**. The router is only as good as its training data. Assembling a dataset of queries with reliable optimal-weight labels is time-consuming and often requires manual annotation or careful inference from user feedback signals. Sparse or biased training data will produce a router that fails on the query types it hasn't seen—which are often the queries where adaptive routing matters most.
- **Black-box decision making**. Once trained, the model's routing logic is opaque. When the router produces a suboptimal weighting for a particular query, diagnosing why requires interpretability tooling and ML debugging skills. This lack of transparency makes it harder to explain system behavior to stakeholders, and it complicates the feedback loop when retrieval quality degrades.
- **Ongoing maintenance and retraining**. A learned router is not a "ship and forget" component. As your content library grows, user behavior shifts, or new query patterns emerge, the model's predictions will drift. Maintaining retrieval quality requires continuous monitoring, periodic retraining on fresh data, and regression testing—an operational commitment that compounds over time.
- **Long iteration cycles**. When routing behavior needs to change, you cannot simply update a configuration file. You must collect new training data, retrain the model, validate it against held-out queries, and deploy the updated version. This cycle can take days or weeks, making it difficult to respond quickly to emerging issues or shifting user needs.
- **Latency and infrastructure overhead**. The router adds a model inference step to every query. While small models can run quickly, this still introduces latency and requires inference infrastructure—GPU resources, model serving frameworks, and monitoring. For latency-sensitive applications, this overhead must be carefully managed.

4.2.4 - Recommendations

Best for

- Organizations with **established ML infrastructure and dedicated ML engineering resources**. If your team already operates model training pipelines, serving infrastructure, and monitoring systems, adding a routing model fits within existing operational patterns.
- Large-scale deployments with **stable, well-understood query distributions**. When your query patterns are complex but consistent—and you have sufficient labeled data to capture that complexity—a learned router can deliver routing precision that exceeds rule-based or anchor-based alternatives.
- Use cases where **routing accuracy justifies the investment**. If small improvements in modality weighting translate to meaningful business outcomes (e.g., user engagement, conversion, or compliance), the upfront cost and ongoing maintenance of a learned router may be warranted.
- Teams that can commit to **long-term model maintenanc**e. A learned router is a living system component. Organizations prepared to monitor performance, collect fresh training data, and retrain periodically will see sustained value; those expecting a one-time investment will not.

Not ideal for

- Teams **without ML expertise or infrastructure**. If building, training, and maintaining a model is outside your core competency, the overhead will outweigh the benefits. Anchor-based routing (Section 4.3) or statistically optimized fixed weights (Section 4.1) offer adaptive or tuned behavior without the ML burden.
- Early-stage projects where **query patterns are still evolving**. A learned router trained on today's query distribution may not generalize to tomorrow's users. If you are still discovering how users interact with your system, investing in a trained model is premature.
- Deployments that require **transparency and explainability** in routing decisions. If stakeholders need to understand why the system routed a query a certain way—for compliance, debugging, or user trust—the black-box nature of a learned model is a liability.
- Organizations that need **rapid iteration** on routing behavior. If your routing logic needs to change frequently in response to user feedback or content shifts, the long retrain-and-deploy cycle of a learned router becomes a bottleneck. Configuration-based approaches allow faster adaptation.

Generally, **a specialized ML router is the right choice when you have the data, expertise, and operational commitment to treat routing as a first-class ML problem**. It offers the highest ceiling for routing precision—but only for teams prepared to invest in building and maintaining it over time. For most organizations, anchor-based routing (Section 4.3) delivers comparable adaptivity with significantly lower overhead.

4.3 - Intent-based Dynamic Query Routing

4.3.1 - Overview

The key problem with dynamic routing isn't "how do we merge results?"—it's "how do we decide which modality should matter for this query, in a way that is deterministic and debuggable?"

This approach uses **embeddings themselves** to answer that question.

Instead of training a specialized ML model as the router (Section 4.2) or asking an LLM to decompose the query (Section 3.2.2), the system constructs a small set of **routing "anchors"** that represent modality intent (e.g., "visual content," "non-speech audio," and "spoken dialogue"), then measures how closely the query aligns with each anchor. Those similarities become routing signals.

4.3.2 - Technical Implementation

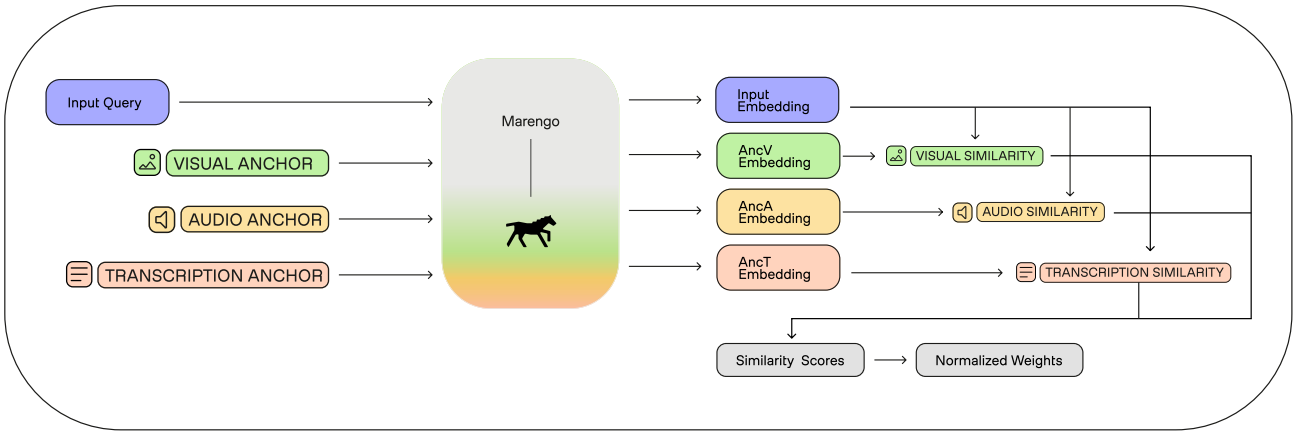


Figure 7. Diagram representing the use of routing anchors to create dynamic routing weights based on the input query.

The implementation is straightforward once you understand the core intuition: use the embedding space itself to measure how closely a query aligns with each modality intent, then convert those alignment scores into routing weights. The process follows three steps:

1. Prepare routing "anchors" per modality. Anchors represent the semantic intent of each modality and can be created in a number of ways. The most straightforward method for creating these modality anchors is to create a representative text description of each modality. Anchors can be as simple as these example anchors:
   Visual Anchor (AncV): "This document contains content about the visual elements of the video."

   Audio Anchor (AncA): "This document contains content about the audio elements of the video."

   Transcript Anchor (AncT): "This document contains content about the spoken words in the audio elements of the video."

2. Embed the input query and the routing anchors using Marengo.

3. Calculate the cosine similarity scores between the query embedding and the anchor embeddings, then normalize the scores into weights based on the formula below:

$$(w_v,\ w_a,\ w_t) = \sigma(\alpha \cdot sim(E_{query},\ (E_{AncV},\ E_{AncA},\ E_{AncT})))$$

Equation 4. Formula for calculating the modality weights based on routing anchors and an input query.

Where:

- w_v, w_a and w_t represent the final weights for the visual, audio, and transcription embedding modalities respectively.
- E_{query} is the Marengo embedding of the input query.
- E_{AncV}, E_{AncA}, and E_{AncT} represent the embeddings of each routing anchor, embedded by Marengo.
- $sim(A, B)$ represents the cosine similarities between vector A and a list of vectors B.
- $\sigma(C)$ represents the softmax (a.k.a. softargmax or normalized exponential function) of between the values in a tuple C. The softmax function converts a tuple of n real numbers into a probability distribution over n possible outcomes (in this case, $n = 3$).
- α represents the temperature, where an $\alpha > 1$ amplifies and an $\alpha < 1$ dampens the differences between the input values of the softmax function. Since cosine similarity outputs a max value of 1, we recommend setting $\alpha = 10$ as the default to amplify the modality with a higher similarity score.

4.3.3 - Advantages

Deriving routing signals from embedding similarity to anchor intents has four practical advantages:

- **Deterministic behavior.** The same query produces the same routing weights every time, given the same embedding model and anchor configuration. This repeatability is essential for production systems where inconsistent behavior erodes user trust and complicates issue resolution. When a user reports a relevance problem, you can reproduce the exact retrieval path—eliminating the "it worked differently last time" ambiguity that plagues non-deterministic approaches.
- **Explainability.** Routing decisions are not just reproducible—they are interpretable. You can inspect the similarity scores between the query and each anchor, revealing which modality alignment drove the weighting and by how much. This provides actionable signal when tuning retrieval quality: if transcription-heavy queries are underperforming, you know exactly where to look. It also enables meaningful conversations with non-technical stakeholders who need to understand why the system behaves the way it does.
- **Configurable without ML model training.** The entire routing logic executes at query time using precomputed anchor embeddings, which means updates can be deployed without retraining models or re-indexing content. Each modality anchor can be revised independently—adjusting the transcription anchor, for example, does not require touching the visual or audio anchors. This modularity enables fast, low-risk iteration as you learn more about your query distribution.
- **Extensible architecture.** The anchor-based routing pattern establishes a foundation that grows with your retrieval needs. Adding new routing dimensions (e.g., a metadata or entity-focused anchor) requires only defining the new anchor and incorporating it into the softmax calculation—not redesigning the pipeline. The approach can also be layered with LLM-based query decomposition (Section 3.2.2) to handle complex, multi-part queries—using anchors for routing while relying on the LLM to parse conditional or compound intent. This extensibility makes the design well-suited for enterprise systems that evolve toward hybrid retrieval over time.

4.3.4 - Drawbacks

Intent-based dynamic query routing avoids the black-box nature of learned routers and the non-determinism of LLM-generated weights, but it introduces its own design and operational considerations.

- **Anchor design is a new engineering surface.** The quality of routing depends heavily on how well your anchors capture modality intent. Poorly designed anchors—too generic, too specific, or semantically overlapping—will produce routing weights that do not align with actual query intent. This requires iteration and domain knowledge to get right.
- **Temperature parameter requires tuning.** The sharpness of the softmax distribution—controlled by the temperature parameter α—affects how decisively the system routes toward a dominant modality versus spreading weight across all three. Suboptimal temperature settings can produce routing that is either too aggressive (ignoring relevant modalities) or too diffuse (diluting the benefit of dynamic routing).
- **Limited expressiveness for complex queries.** Anchor-based routing captures broad modality intent but may struggle with queries that have nuanced, conditional structure (e.g., "find the moment where X is visible but only if Y is being said"). For such cases, layering with LLM-based query decomposition (Section 3.2.2) may be necessary to fully capture user intent.

4.3.5 - Recommendations

Best for

- Teams that need **query-adaptive routing without the overhead of training and maintaining a specialized ML model**. If you want dynamic behavior but lack the ML infrastructure or expertise to build a learned router, anchor-based routing provides a principled alternative.
- Deployments where **determinism and auditability are requirements.** Regulated industries, enterprise customers with strict SLAs, and applications where users expect consistent behavior will benefit from routing logic that is fully reproducible and inspectable.
- Organizations that expect **ongoing iteration on routing behavior.** Because anchors can be updated independently and without retraining, this approach suits teams that want to refine routing as they learn more about their query distribution—without incurring costly re-deployment cycles.
- Use cases with **reasonably well-defined modality intent boundaries.** When queries tend to fall into recognizable categories—primarily visual, primarily dialogue, primarily audio-event—anchor-based routing can capture that structure effectively.

Not ideal for

- Teams that need **immediate, out-of-the-box routing** without investing in anchor design and tuning. If you lack the time or domain knowledge to craft effective anchors, fixed-weight approaches (Section 4.1) will deliver results faster with less upfront effort.
- Use cases dominated by **complex, conditional queries** that require parsing beyond modality intent (e.g., "find X only when Y is also true"). For these queries, anchor-based routing alone may be insufficient—consider layering with LLM-based query decomposition for richer intent handling.

Generally, intent-based dynamic query routing is the right choice when you need adaptive, per-query weighting but want to avoid the black-box nature of learned models and the non-determinism of LLM-generated weights. It occupies a practical middle ground: more responsive than fixed weights, more predictable than fully generative approaches.

5 - Conclusion

5.1 - Comparative Summary of Approaches

This whitepaper has covered two foundational architectures—fused embeddings and multi-vector retrieval—along with three strategies for determining modality weights in multi-vector systems. Each approach occupies a different point on the spectrum of simplicity versus adaptivity, and the right choice depends on your query distribution, operational constraints, and team capabilities.

Approach	Index Design	Query Adaptivity	Required Expertise	Pros	Cons	Best Fit
Fused Embeddings (Section 2)	Single index (one fused vector per segment)	None (weights fixed at ingestion)	Minimal; familiar RAG patterns	Simplest implementation; single retrieval pass; lowest operational overhead	Fixed weights baked into embeddings; limited debuggability; irreversible fusion	Quick start; visually-skewed query distributions; teams prioritizing simplicity
Multi-Vector with Fixed Weights (Section 3 + 4.1)	Three indices (visual / audio / transcription	None (global fixed weights at query time)	Data science capability for statistical optimization	Modality separation preserves debuggability; weights can be tuned without re-indexing; supports modality-specific scaling; stepping stone to dynamic routing	Static weights do not adapt to query intent; increased storage footprint and query latency; score calibration across modalities is non-trivial; requires evaluation dataset and optimization workflow	Teams needing transparency and tuning; stable query distributions with known modality skew
Multi-Vector with Specialized ML Router (Section 3 + 4.2)	Three indices + learned routing model	Yes (per-query weights from trained model)	Dedicated ML team; training and serving infrastructure; ongoing maintenance capacity	True query-adaptive behavior; can learn domain-specific patterns; single inference component in deployment	Black-box decisions; significant data labeling burden; long iteration cycles for updates; ongoing retraining required; adds inference latency	Large-scale deployments with stable patterns; organizations with mature ML infrastructure and long-term maintenance commitment
Multi-Vector with Intent-Based Dynamic Routing (Section 3 + 4.3)	Three indices + anchor-based routing	Yes (per-query weights from anchor similarity)	Moderate; requires anchor design and tuning	Deterministic and explainable; configurable without ML training; extensible architecture; can layer with LLM decomposition for complex queries	Anchor quality determines routing quality; adds query-time compute; temperature parameter requires tuning; may not generalize across diverse content domains	Teams needing adaptivity without ML overhead; deployments requiring auditability and rapid iteration

5.1.1 Reading the Trade-offs

The progression from fused embeddings to dynamic routing represents a trade-off between simplicity and **retrieval precision across diverse queries.**

Fused embeddings offer the fastest path to production. If your queries are predominantly visual and your priority is operational simplicity, fusion delivers sufficient quality with minimal infrastructure. However, the weighting decision is permanent—changing it means reprocessing your entire corpus.

Multi-vector retrieval with fixed weights introduces modality separation, which enables debugging and tuning without the irreversibility of fusion. Statistical optimization can improve upon default weights, but the system still applies the same weighting to every query. This works well when query intent is stable and predictable.

Multi-vector retrieval with a specialized ML router provides true per-query adaptivity, but at significant cost: you need labeled training data, ML expertise, and a commitment to ongoing model maintenance. The routing logic is powerful but opaque, which complicates debugging and stakeholder communication.

Multi-vector retrieval with intent-based dynamic routing offers a middle path: per-query adaptivity with deterministic, explainable behavior. Anchor-based routing avoids the black-box nature of learned models and the non-determinism of LLM-generated weights, while still responding to query semantics. The trade-off is that routing quality depends on anchor design, which requires iteration and domain knowledge.

5.1.2 Choosing the Right Approach

For most teams, the practical decision comes down to two questions:

1. How diverse is your query distribution? If most queries favor similar modality balances, static approaches (fused or fixed-weight multi-vector) are likely sufficient. If query intent varies significantly—spanning visual, dialogue, and audio-event queries—dynamic routing becomes the higher-ROI investment.
2. What expertise and operational capacity does your team have? Fused embeddings require minimal specialized skill. Fixed-weight optimization requires data science capability. Learned routers require ML engineering and infrastructure. Anchor-based routing requires thoughtful design but avoids the ML maintenance burden.

5.2 - Best Practices

The approaches in this whitepaper are not mutually exclusive. Production systems often combine multiple strategies—layering anchor-based routing with LLM query decomposition, or using fixed weights for simple queries while routing complex queries dynamically. The key is understanding how each decision affects four critical dimensions: time to value, operational complexity, search performance, and transparency.

5.2.1 Balancing Time to Value

Start simple and evolve incrementally.

- Begin with the simplest approach that delivers acceptable retrieval quality—typically fused embeddings or multi-vector retrieval with default weights.
- Use your baseline to gather real query data, user feedback, and performance signals before investing in more sophisticated approaches.

- A working system with fixed weights will teach you more about your query distribution in one month of production traffic than six months of theoretical planning.

Design for incremental upgrades.

- The progression from fused embeddings to multi-vector retrieval to dynamic routing is designed to be incremental; each step builds on the infrastructure of the previous one.
- Avoid architectures that require full re-implementation to improve. Prefer designs where adding dynamic routing or LLM decomposition is an additive change, not a replacement.

5.2.2 Managing Operational Complexity

Understand where complexity accumulates.

- Index management: Multi-vector retrieval requires three indices instead of one—more storage, more write paths, more failure surfaces.
- Query-time orchestration: Dynamic routing adds logic that must be monitored and maintained. LLM-based decomposition introduces external dependencies and latency.
- Model maintenance: A specialized ML router requires training infrastructure, labeled data pipelines, and ongoing retraining cycles.

Be deliberate about where you take on complexity.

- Ensure the retrieval quality gains justify the operational burden for each added component.
- When combining approaches (e.g., anchor-based routing with LLM decomposition), consider the compounding effect on operational overhead.
- Design for scale from the start: consider how you will shard indices, manage replication, and meet latency SLAs as your corpus and query volume grow.

5.2.3 Maximizing Search Performance

Match your architecture to your query distribution.

- Analyze your query logs (or anticipated query patterns) to understand the mix of modality intents.
- If most queries favor similar modality balances, static approaches will serve you well.
- If query intent varies significantly across visual, dialogue, and audio-event queries, dynamic routing becomes essential.

Invest in evaluation infrastructure early.

- You cannot improve what you cannot measure.
- Assemble a representative query set, define ground truth relevance labels, and select metrics that align with your application's success criteria.
- Evaluation infrastructure pays dividends at every stage: it tells you whether your current approach is sufficient and whether proposed improvements actually improve relevance.

Combine strategies where their strengths complement each other.

- Anchor-based routing can provide fast, deterministic modality weighting for most queries, while LLM decomposition is selectively invoked for complex, multi-part queries.
- Statistically optimized fixed weights can serve as a fallback when dynamic routing produces low-confidence results.
- Layer approaches intentionally rather than choosing a single strategy for all query types.

5.2.4 Ensuring Transparency and Control

Preserve modality separation for debuggability.

- When retrieval quality degrades, you need to diagnose the cause quickly.
- Architectures that preserve modality separation let you inspect whether a result matched on visual, audio, or transcription signals.
- Systems that fuse or obscure modality contributions make debugging harder and slower.

Design for explainability when it matters.

- If your application serves enterprise customers or operates in regulated environments, you may need to explain why the system returned specific results.
- Black-box components—whether fused embeddings or learned routers—make this difficult.
- Anchor-based routing and explicit score-based fusion provide audit trails that can answer "why did this result rank here?"

Preserve optionality where practical.

- Even if you start with fused embeddings, consider storing the original modality embeddings alongside the fused representation.
- Fusion is irreversible; upgrading to multi-vector retrieval later means reprocessing your entire corpus if you haven't retained the separate embeddings.
- The storage cost of keeping separate embeddings is modest compared to the reprocessing cost of not having them.

Document your decisions.

- The choices you make—fusion weights, anchor text, optimization metrics, routing thresholds—encode assumptions about your content and users.
- When those assumptions change, or when a team member needs to understand system behavior, the rationale should be discoverable.

5.3 - Closing Thoughts

Video is not a single signal—it is a synchronized bundle of visual, auditory, and spoken information. The retrieval systems that treat it as such will consistently outperform those that collapse this richness into a single undifferentiated representation.

This whitepaper has provided a framework for operationalizing multi-vector embeddings in production: how to choose between fused and separated architectures, how to determine modality weights (whether fixed, statistically optimized, or dynamically routed), and how to combine strategies where their strengths complement each other. The right approach depends on your query distribution, team capabilities, and operational constraints—but the underlying principle remains constant: **preserve the structure of the source medium, and let user intent guide how that structure is leveraged at query time.**

5.3.1 Beyond Multi-Vector Search

Multi-vector search is rarely the complete retrieval solution for enterprise deployments. Production systems often combine semantic multi-vector search with structured metadata filtering—restricting results by date range, content type, speaker identity, geographic region, or other cataloged attributes. Hybrid architectures that blend dense vector similarity with sparse keyword matching or metadata constraints can deliver retrieval experiences that neither approach achieves alone.

These hybrid search strategies—and how they integrate with the multi-vector approaches covered here—will be the subject of a forthcoming companion whitepaper.

5.3.2 A Foundation for What Comes Next

The architectures described in this whitepaper are not endpoints. As embedding models improve, as query patterns evolve, and as enterprise requirements grow more sophisticated, the systems built on these foundations will need to adapt. The teams that succeed will be those that design for evolution: preserving modality separation, investing in evaluation infrastructure, documenting their decisions, and treating retrieval as a capability that improves over time rather than a problem to be solved once. Marengo provides the embedding foundation. The rest—how you index, how you route, how you rank, and how you learn from user feedback—is where differentiated retrieval experiences are built.

TwelveLabs

TwelveLabs delivers industry-leading video AI solutions that unlock the full potential of vast video libraries. Our proprietary multimodal foundation models bring human-like understanding to videos, enabling precise semantic search, summarization, analysis and Q&A through easy-to-integrate APIs. This empowers enterprises to effortlessly search, monetize IP, extract insights, and repurpose content at scale.

Unlike conventional methods that struggle with video; TwelveLabs overcomes the limitations of manual tagging and inadequate computer vision techniques, streamlining processes with customizable models. These models make previously inaccessible video assets searchable, seamlessly integrating into existing workflows. From industry leaders in media and entertainment to governments around the world TwelveLabs is changing the way the world works with video.

Website: twelvelabs.io
Documentation: docs.twelvelabs.io
Blog: twelvelabs.io/blog
Follow Us: LinkedIn | X
Contact Us: sales@twelvelabs.io

TwelveLabs

twelvelabs.io

