Cost-Efficient Multimodal Ingestion Architecture for K-12 EdTech Platforms

A Tiered Approach to Model Routing, Storage, and Structured Signal Extraction

March 2026

Abstract:

A practical architecture for ingesting and processing multimodal student artifacts (text, PDFs, audio, video, images) at K-12 scale. The approach prioritizes cost efficiency through a tiered extraction strategy; that is, using deterministic tools first, small language models by default, and large models only as fallback. We present a unified JSON schema that abstracts source modality from downstream features, a PostgreSQL + pgvector storage design enabling hybrid search with row-level multi-tenant isolation, and a model routing strategy that keeps per-artifact processing costs under two cents while maintaining quality gates.

Author:
Omer Ashfaq
omer@omerashfaq.com
Github.com/yunomer
Linkedin.com/in/omerashfaq

Introduction

This paper presents the design of a minimal, practical AI pipeline for processing user-generated artifacts in a K-12 educational platform. The objective is to transform heterogeneous inputs into a unified structured output that powers downstream features including automated feedback, longitudinal user insights, and semantic search. Our approach prioritizes cost efficiency at scale, targeting per-artifact processing costs in the fraction-of-a-cent range while maintaining sub-15-second end-to-end latency. We make the case that intelligent model routing, tiered extraction strategies, and consolidated storage using PostgreSQL with pgvector can meet these constraints without the operational overhead of a multi-database architecture. The following sections detail the ingestion pipeline, data schema, storage approach, and routing logic, concluding with an end-to-end walkthrough of a real processing flow.

Multimodal Ingestion Pipeline

My starting point for this pipeline is a simple rule: *don't call a model if you don't have to*. For every artifact type, I'd try the cheapest, fastest extraction method first, and only escalate to an AI model if that fails. At K-12 margins with a million students on the platform, per-artifact cost has to stay at fractions of a cent, and this principle is how I'd get there.

For text posts, there's really nothing to extract. The text is already machine-readable, so I'd just sanitize the input (regex cleaning, personally identifiable information detection (`PII`)), count words, detect the language, and pass it straight through to signal extraction. No model involved. Cost is effectively zero, latency under 50 milliseconds.

For PDFs, I'd take a two-tier approach. Most student PDFs are born-digital (exported from Google Docs or Word), so I'd run them through PyMuPDF first. It's a deterministic parser, no AI, and it extracts text locally in about 120 milliseconds at zero API cost. But if PyMuPDF comes back with little or no text, that probably means it's a scanned document. Only then would I fall back to a cheap SLM e.g. GPT-4o-mini vision, at about $0.001 per page. The reason I like this split is that 90%+ of student PDFs cost nothing to extract, and I'm only paying for the genuinely hard ones.

For audio and video, I'd use the OpenAI Whisper API for transcription. If there's a video track, I'd strip the audio with FFmpeg first (no point sending pixels to a transcription service). Whisper runs $0.006 per minute, so a typical 3-minute student reflection costs under two cents. It handles 99 languages with automatic detection and hits 95-97% accuracy on clear audio. The main limitation is that it's batch-only, no real-time streaming, but for recorded reflections that doesn't matter. One thing I'd flag is if privacy requirements are strict enough that audio can't leave your infrastructure, self-hosted Whisper is a viable alternative. Same model, you just run the GPU yourself on local infrastructure. Nvidia blackwell GPU chips are relatively cheap and can be deployed locally.

Images are the modality I thought about the most. When a student uploads a photo of their handwritten lab notes, or their art project, or a worksheet, you don't just want the characters on the page. You want the system to understand what the student is showing you. I considered traditional OCR tools like Tesseract, but they only extract text. GPT-4o-mini in vision mode does both: it pulls out readable text and generates a description of the visual content ("A hand-drawn diagram of the water cycle with labeled arrows"). That dual output is much more useful when you're trying to generate feedback later. The tradeoff is latency: 1-3 seconds vs Tesseract's 200 milliseconds. But since I'd run all ingestion asynchronously, the student never sits there waiting, so I think it's worth it.

Here's how the whole pipeline fits together: Appendix 1

Downstream AI use cases

I won't design these features here, but I want to lay out what each one needs from the structured output, since that's what drove my schema decisions.

The Feedback Assistant generates short, actionable feedback on a single artifact. For that, it needs the extracted text, detected topics, quality signals, and artifact type. The User Insights Summary is a different beast of a problem: it looks across many artifacts to find recurring themes, so it needs fields that are consistent and comparable from one artifact to the next. Search and Retrieval needs a dense vector embedding for semantic similarity, plus filterable metadata like date, type, and subject so you can do hybrid queries.

Structured output schema

The core idea behind my schema is that every artifact, regardless of whether it started as audio, an image, or a PDF, gets transformed into the same JSON structure. I did this with intention: downstream features shouldn't need to know or care what the original input format was. They just read from one shared schema (Appendix 2).

I'll walk through my reasoning for each section:

The `extraction` block is the raw material that every downstream feature reads from. The `extracted_text` and `visual_description` feed into the Feedback Assistant, get aggregated by User Insights, and get indexed by Search. I included `extraction_confidence` because VLMs can hallucinate, especially on low-quality images. If that score is low, I'd want the system to flag the artifact for human review rather than letting automation run on unreliable data.

The `signals` block is where the SLM's structured analysis lives. I designed topics to power recurring-theme detection in User Insights and faceted filtering in Search. The `summary` anchors feedback generation and is itself searchable. I put `strengths` and `quality_indicators` in mainly for the Feedback Assistant (so it can calibrate its tone) and User Insights (so it can track growth, like noticing a student has been consistently strong at "visual communication" across their last ten artifacts).

One design decision I want to call out is there are two separate confidence fields. I split them because extraction can succeed (the text is correct) while signal analysis might be uncertain (the detected topics are ambiguous). They gate different things. Low extraction confidence means "don't trust the text." Low signal confidence means "maybe retry this with a bigger model."

I also added `input_text_hash` on the embedding as a small optimization: if a student updates an artifact but the underlying text hasn't actually changed, the system skips re-embedding.

Storage and retrieval approach

I'd store everything in two systems, not four.

For raw files (images, PDFs, audio, video), I'd use S3. It costs $0.023 per gigabyte per month, has eleven nines of durability, and you can set lifecycle policies to archive old files automatically. I'd organize files at `/{tenant_id}/{user_id}/{artifact_id}/`, which lets you enforce tenant-level access through IAM policies before any application code gets involved.

For everything else (metadata, extracted text, structured signals, embeddings), I'd put it all in PostgreSQL with the pgvector extension. You might expect me to recommend a separate vector database here, like Pinecone or Qdrant. I considered them, but I'd argue against it for this use case. The typical RAG architecture pairs a relational database with a dedicated vector store. That works, but it creates a synchronization problem: two databases to keep in sync, failures to handle in either one, two sets of backups and monitoring to maintain. For a small EdTech team, that operational overhead matters more than it might seem.

What I like about pgvector is that embeddings live in the same table, same row, as the `student_id` and `tenant_id` and all the structured signals. The biggest win, and the reason I'd go this route, is multi-tenant isolation through PostgreSQL Row-Level Security. Every query is automatically scoped to the current tenant at the database level, not the application level. For a platform handling Canadian student data under PIPEDA, the difference between "the app filters by tenant" and "the database enforces it" is quite significant. You also get hybrid search in a single SQL query: vector similarity (cosine distance on the embedding column), full-text search (tsvector on `extracted_text`), and relational filters (*WHERE modality = 'image' AND created_at > ...*) all in one statement. Try doing that across two separate databases and you're writing application-level joins, which are slower and more fragile.

The tradeoff: pgvector won't outperform a dedicated vector database at truly massive scale. But pgvectorscale benchmarks at 471 queries per second at 99% recall on 50 million vectors. Even if every student on the platform uploads 100 artifacts, that's ***100 million rows***, which is within range. If we ever needed to go past that, migrating embeddings to a dedicated store would make sense. But for K-12 scale, I'd rather have one database that does everything than deal with the operational cost of a specialized tool.

To give a concrete sense of what retrieval looks like:

```
SELECT * FROM artifacts WHERE tenant_id = $1 ORDER BY embedding <=> $query_vector LIMIT 10.
```

Single query, tenant-scoped, would have a sub-10 millisecond response at moderate scale.

Model routing strategy

My approach to routing comes down to: rules first, small models by default, big models by exception (Appendix 4).

Before any model sees any content, I'd run a layer of heuristic checks. They cost nothing and take under 10 milliseconds. If the text posts are under 5 words, return a prompt asking for more detail, no model needed. If the PDF has extractable text, use PyMuPDF deterministically. If there is PII in the text, redact before it reaches a model. Does the MIME type tell you which pipeline to route to? These checks eliminate a real number of unnecessary model calls, and every call you skip is money saved.

For everything that actually needs a model, I'd default to a small language model. This would handle 80-90% of all AI work. Models like GPT-5.4 nano (about $0.20 per million input tokens), Claude 4.5 Haiku, or Gemini 3.1 Flash can handle everything that involves a single artifact: extracting structured JSON, generating feedback, summarizing a document, image OCR, classification. They're 10-20x cheaper than frontier LLMs and 2-5x faster. To put a number on it: processing one million student artifacts per month at roughly 500 tokens each costs about $75 with GPT-5.4 nano. The same volume with GPT-5 would cost about $1,250. That's the difference I keep coming back to when thinking about routing.

I'd only reach for a large language model for the hardest 10-20% of tasks. The clearest example is the User Insights Summary, which needs to synthesize patterns across dozens of artifacts. That kind of

long-context, multi-document reasoning is where bigger models still have a real edge over SLMs. The other situation where I'd use an LLM is as a fallback is if an SLM's output fails JSON schema validation, or its confidence score comes back below 0.7, the system retries with a more capable model and passes along the SLM's failed attempt as context so the LLM has a head start.

Where I'd actively avoid using an LLM would be single-artifact extraction or feedback (the SLM handles it fine), anything a deterministic tool can do (PDF parsing, file type detection), and any high-volume repetitive task where cost scales linearly with volume (like processing every new upload).

I'd also want the routing layer to track SLM success rates per task type over time. If the SLM handles more than 95% of feedback requests correctly, it stays at that tier. If it drops below 80% on some task, that task type gets escalated to the LLM. This gives you a self-adjusting cost loop that doesn't require someone to manually tune thresholds.

End-to-end example flow

Let me walk through a concrete example to show how all of this connects.

1. A grade 8 student records a 3-minute audio reflection about their science fair project and uploads it. The upload service writes the raw .m4a file to S3 at `/{tenant_id}/{user_id}/{artifact_id}/reflection.m4a` and drops an async processing job onto the queue.

2. The ingestion worker picks up the job and checks the `MIME type`. It's audio, so it routes to the audio pipeline. No video track to strip, so FFmpeg gets skipped. The file goes to the Whisper API, which returns a transcript in about 8 seconds for $0.018: *"For my science fair project I built a model volcano. I used baking soda and vinegar for the eruption. I learned about chemical reactions and how gases expand when heated..."*

3. The transcript, plus whatever the student typed as a caption, goes to GPT-5.4 nano with a structured output prompt. In about 400 milliseconds, it comes back with JSON containing a summary, detected topics (chemical reactions, science fair, volcanoes), strengths (hands-on experimentation, cause-and-effect reasoning), and quality indicators. Signal confidence is 0.85 and the output passes schema validation, so there's no need to escalate to a larger model.

4. The summary and extracted text then go to text-embedding-3-small, which returns a 1536-dimensional vector. Everything (metadata, transcript, structured signals, embedding) gets written to a single row in the PostgreSQL artifacts table, scoped to the student's tenant via RLS.

5. At this point, the artifact is ready for all three downstream features. A teacher searching for "chemical reactions" will find it through semantic similarity against the embedding. The Feedback Assistant can write an encouraging note about the student's experimental reasoning. And the next time User Insights runs for this student, it'll notice that "science" and "hands-on experimentation" keep showing up across their portfolio.

Total processing cost: about two cents. Wall-clock time: roughly 12 seconds, all in the background! The student never waits for any of it.

Appendix 1:

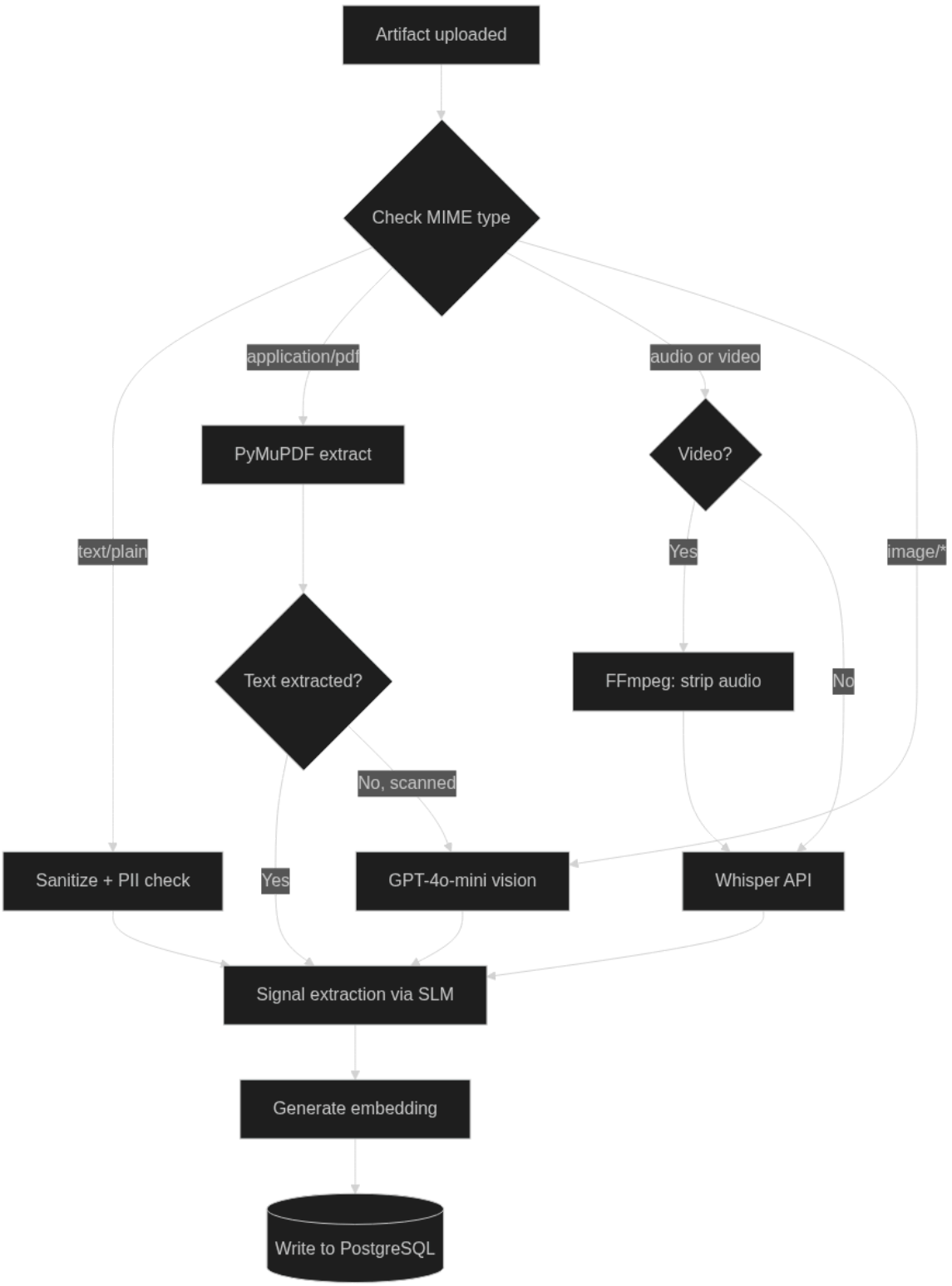

Figure: Proposed multimodal ingestion pipeline flowchart

Appendix 2:

```json
{
  "artifact_id": "a1b2c3d4",
  "user_id": "u9876",
  "tenant_id": "district-42",
  "created_at": "2025-03-15T10:30:00Z",
  "processed_at": "2025-03-15T10:30:12Z",

  "source": {
    "modality": "image",
    "mime_type": "image/jpeg",
    "raw_storage_uri": "s3://artifacts/district-42/u9876/a1b2c3d4/photo.jpg",
    "user_provided_text": "My water cycle diagram for science class"
  },

  "extraction": {
    "extracted_text": "Evaporation, Condensation, Precipitation, Collection...",
    "visual_description": "A hand-drawn diagram of the water cycle with labeled arrows...",
    "word_count": 47,
    "language": "en",
    "extraction_method": "gpt-4o-mini-vision",
    "extraction_confidence": 0.88
  },

  "signals": {
    "summary": "A diagram of the water cycle showing understanding of evaporation, condensation, precipitation, and collection.",
    "topics": ["water cycle", "earth science", "evaporation", "condensation"],
    "strengths": ["visual communication", "scientific labeling", "process understanding"],
    "quality_indicators": {
      "depth": "moderate",
      "completeness": "high",
      "effort_level": "high"
    },
    "signal_confidence": 0.82,
    "model_used": "gpt-4o-mini"
  },

  "embedding": {
    "model": "text-embedding-3-small",
    "dimensions": 1536,
    "input_text_hash": "sha256:abc123..."
  }
}
```

Figure: Proposed output schema

Appendix 3:

Figure: Proposed two system storage

Appendix 4:

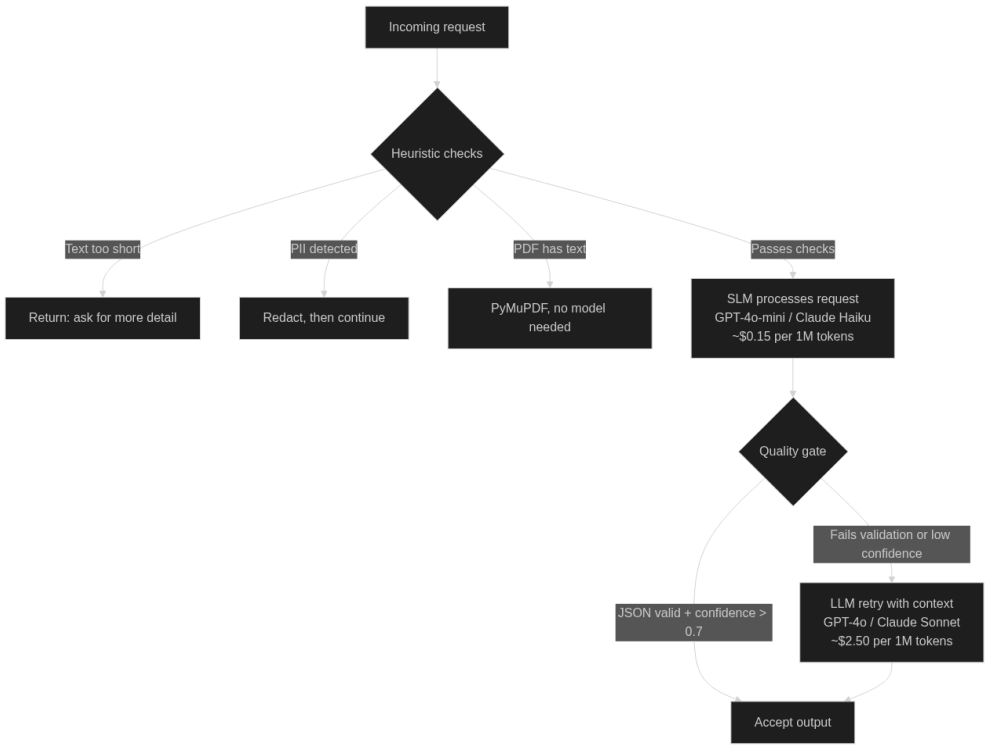

Figure: Workflow diagram illustrating my approach to routing: rules first, small models by default, big models by exception.