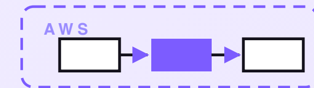
7-PART SERIES · FREE COMPANION



Transcription archive

A serverless archive that keeps every call and meeting recording your business makes, turns each one into text, files it by date, people, and topic, and lets you ask a plain-language question — “find where we discussed the Acme contract” — and get the exact moment back with a quote and a link to play it. Staff still own every decision; the archive just makes the record findable. Access is controlled and every search is logged. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

Workflow guide $19 · Deployable AWS CDK starter $79 · Bundle $89

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/transcription-archive

CONTENTS

Transcription archive

01 A transcription archive on AWS for a few dollars a month

02 How a recording gets filed

03 How a transcript becomes searchable

04 How a question finds the moment

05 How the archive stays private

06 What the transcription archive costs

07 Engineering reference: the transcription archive architecture

PART 1 OF 7

JUNE 7, 2026 PART 1 OF 7 · TRANSCRIPTION ARCHIVE SERIES ~5 MIN READ

A transcription archive on AWS for a few dollars a month

A small business records more than anyone remembers. The sales call where the customer named the real budget. The kickoff where someone promised a delivery date. The support call that became a complaint three weeks later. The all-hands where a policy changed. The recordings pile up in a meeting tool nobody opens, named “Zoom_2026-04-02_1100” and impossible to search. This post walks through the design of a small archive that keeps all of it, turns each recording into text, files it by date, people, and topic, and lets anyone ask “where did we discuss the Acme contract?” and get the exact moment back with a quote.

KEY TAKEAWAYS

- Three sources for recordings: a watched Drive folder, a forward-to-an-address lane, and a scheduled pull from your meeting tool.
- Every recording is transcribed once, filed by date, people, and topic, and split into short timed chunks for search.
- Ask a plain-language question; the closest chunks come back with timestamps and a quote linked to the second in the audio.
- Each recording carries an access tag; search only returns what the asker is allowed to see, and every search is logged.
- Designed on AWS for about $4/month at typical small-business volume.

The whole system on one page

Before any code, here's the shape of what we're designing.

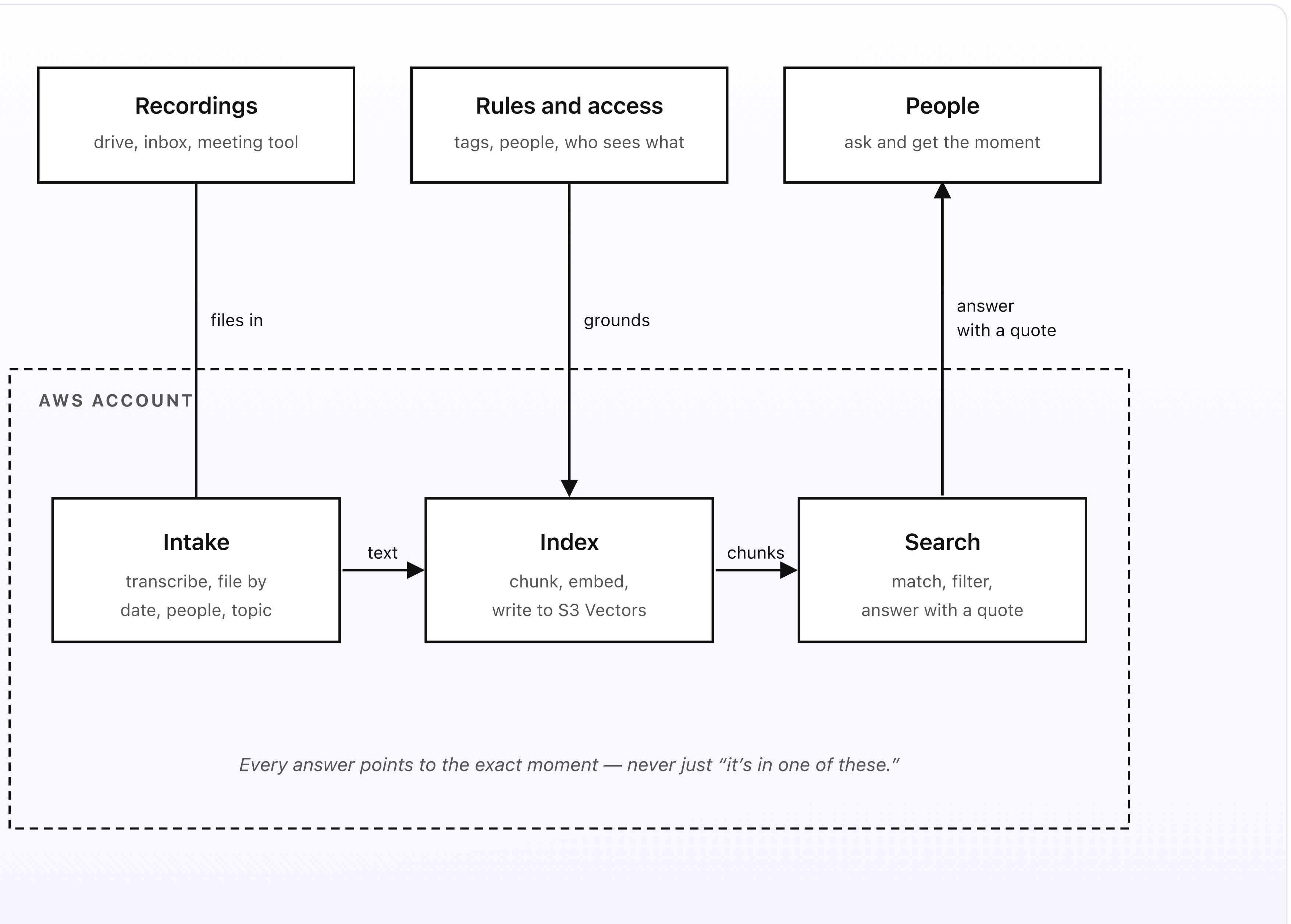

Recordings
drive, inbox, meeting tool
Rules and access
tags, people, who sees what
People
ask and get the moment
files in
grounds
answer
with a quote
AWS ACCOUNT
Intake
transcribe, file by
date, people, topic
text
Index
chunk, embed,
write to S3 Vectors
chunks
Search
match, filter,
answer with a quote
Every answer points to the exact moment — never just “it’s in one of these.”

Fig 1. Three sources outside, three pieces inside AWS. Recordings flow in from a Drive folder, a forwarding lane, and a meeting-tool pull. The Intake transcribes and files. The Index makes it searchable. Search returns the right moment with a quote.

What you set up once (the outside)

- **Recordings.** A Google Drive folder that you drop audio or video into. New recordings can also enter via two other lanes covered in Part 2 — a forward-to-an-address lane (forward a meeting recording to a dedicated address and it gets filed) and a scheduled pull (a small connector grabs finished recordings from your meeting tool's cloud on a timer). Whatever the lane, the file lands in S3, and that's where the work begins.
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the topic tags you care about (deals, support, hiring, product), the people aliases (so "Maria," "M. Santos," and her email all map to one person), and which recordings are open to everyone versus restricted. The *access* doc lists who belongs to which team, so the search box knows who is allowed to see what. Both are plain docs a person can edit without a deploy.
- **People.** The staff who search the archive. Each person types a plain-language question into a small search box — "what did the customer say about pricing on the Acme call?" — and gets back a short answer, a direct quote, the name and date of the recording, and a link that plays the audio from the exact second. They only see results from recordings they're allowed to see.

What runs when a recording arrives (the inside)

- **The intake.** A recording lands in S3 and starts the pipeline. Amazon Transcribe turns the speech into text, with speaker labels and a timestamp on every word.

A plain-Python step then files it: it reads the date from the file, matches the speakers and the “invited” list against the people aliases, and tags a topic from a short keyword pass. The result is one catalogue row per recording in DynamoDB — title, date, people, topic, access tag, and a link to the transcript — plus the full transcript saved in S3.

- **The index.** The transcript is split into short timed chunks (roughly a paragraph each, carrying the start time in the recording). Titan Text Embeddings V2 turns each chunk into a vector — a list of numbers that captures the chunk’s meaning, so two ways of saying the same thing land near each other. The vectors are written to S3 Vectors, AWS’s built-in vector store, each one tagged with the recording it came from and its access tag. This is the one step that makes the back-catalogue searchable by meaning instead of exact words.
- **Search.** Someone types a question. The same Titan model turns the question into a vector, and S3 Vectors returns the closest chunks — but first the search filters out anything the asker isn’t allowed to see. Claude Haiku 4.5 reads the top few allowed chunks and writes a short answer with a direct quote. The result links to the exact second in the audio. Every search writes a row to the search log: who asked, what they asked, what came back, and when.

In plain words

Your sales lead asks: “Didn’t the Acme buyer mention a hard deadline on one of our calls?” Nobody remembers which call. The lead opens the search box and types the question. The archive turns it into a vector, finds the three closest moments across forty recorded calls, drops the two from teams the lead can’t see, and Haiku 4.5 writes: “On the April 2 Acme call, their VP said they needed to

be live before their fiscal year-end on June 30." Below the answer is the quote and a play link that jumps to 14 minutes 22 seconds into that recording. The lead clicks, hears it in the buyer's own voice, and forwards the moment to the account team. The whole thing took fifteen seconds, and the search is logged.

The cost of running this is about $4 a month at SMB volume. The cost of *not* running it is the deadline nobody remembered, the promise made on a call that nobody can find, or the hours a junior spends scrubbing through old recordings to confirm what was actually said.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every answer points to the exact moment — a quote and a play link, not a list of files to scrub.
- Transcribe once, index once. The expensive work happens when a recording arrives, not when someone searches.
- Access is checked before the answer is written. You never see a quote from a recording you can't open.
- Every search is logged — who asked, what they asked, and what came back — for a years-long audit trail.
- The rules and access docs live in Drive. Changing a tag, a person alias, or who-sees-what doesn't need a deploy.
- The archive finds the record. It never decides anything — staff still own the call.

Why this shape

Most teams “keep” their recordings the way they keep junk drawers: everything is technically in there, and nothing can be found. The meeting tool stores the files but names them after timestamps and forgets them after ninety days. Somebody’s notes capture a fraction of what was said, filtered through what they thought mattered at the time. And human memory, of course, fails the moment the person who was on the call goes on holiday or leaves.

The setup above keeps the recordings where they already pile up, but adds a small system that *reads* each one as it arrives, files it cleanly, and makes the whole back-catalogue answer questions in plain language. The expensive work — transcription and indexing — happens once, when the recording lands. Searching is cheap and fast. And because access is checked before any answer is written, the archive can hold sensitive calls without becoming a leak. The system is invisible most days; visible only the moment somebody needs to know what was actually said.

The next four posts walk through each piece in turn: how a recording gets filed, how a transcript becomes searchable, how a question finds the moment, and how the archive stays private. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 7, 2026 PART 2 OF 7 · TRANSCRIPTION ARCHIVE SERIES ~4 MIN READ

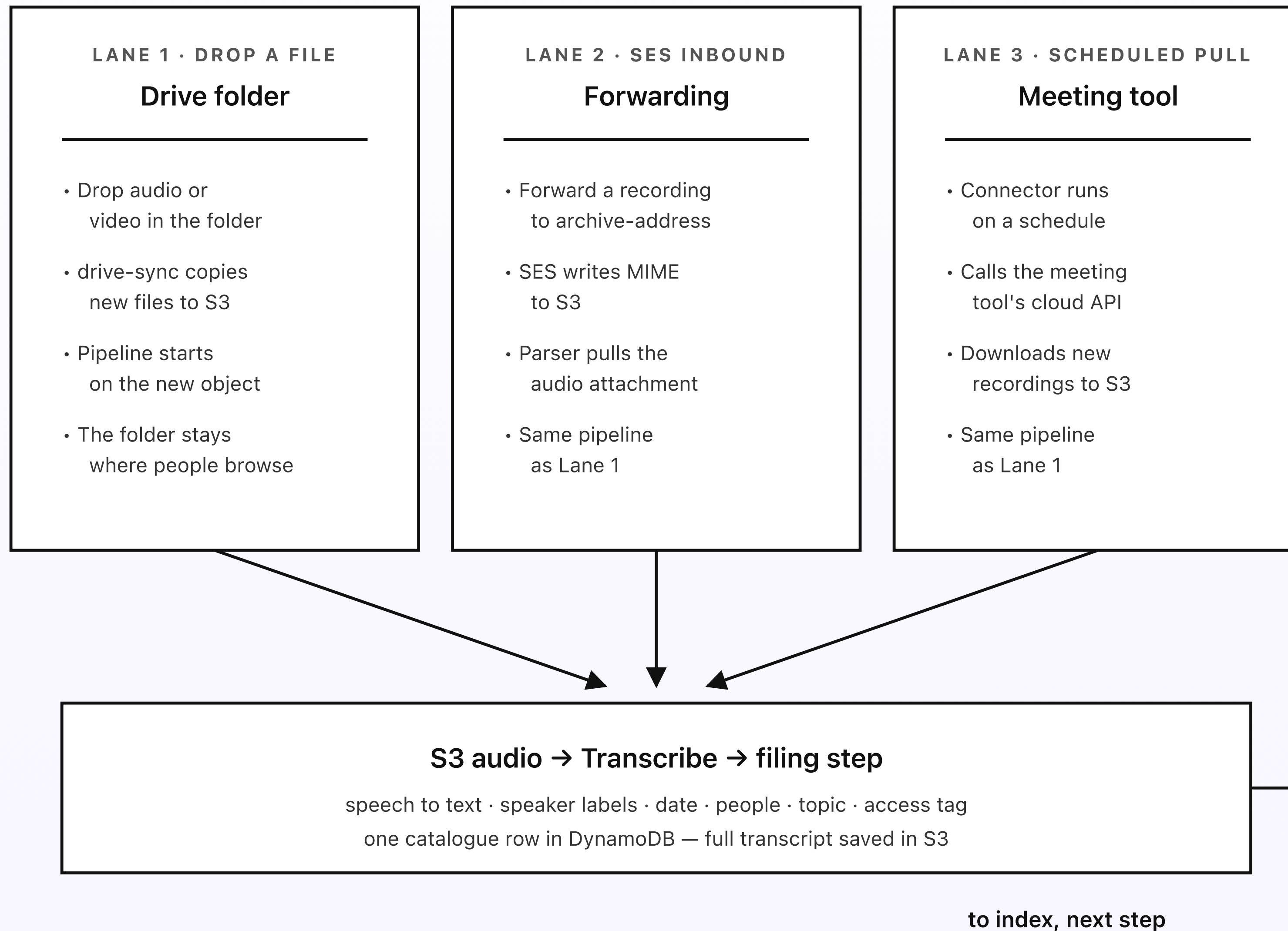
How a recording gets filed

The archive can only search what it has filed. So the first job is making sure recordings actually land in it — without anyone having to remember to upload them. There are three ways a recording gets in: somebody drops a file in a Drive folder, somebody forwards a meeting recording to a dedicated address, or a small connector pulls finished recordings from your meeting tool. Once the file lands, Amazon Transcribe turns the speech into text and a plain-Python step files it by date, people, and topic.

KEY TAKEAWAYS

- Three intake lanes feed one archive: a watched Drive folder, a forwarding lane, and a meeting-tool pull.
- Every lane lands the file in S3, which starts the same filing pipeline.
- Amazon Transcribe turns speech into text with speaker labels and a timestamp on every word.
- A plain-Python step files each recording: date from the file, people from the speakers, topic from a keyword pass.
- The result is one catalogue row in DynamoDB plus the full transcript in S3.

Three lanes into one archive



The file is the source of truth — the three lanes are just different doors into the same pipeline.

Fig 2. Three lanes converge on one S3 bucket. Whatever the door, the file triggers Transcribe and then the filing step. The result is one catalogue row in DynamoDB and the full transcript in S3, ready for indexing in Part 3.

Lane 1: the Drive folder

The simplest lane. Open the watched folder in Drive, drop in an audio or video file, done. A small Lambda — `drive-sync` — runs every few minutes, lists new files in the folder via the Drive API, and copies anything new to `s3://tx-audio/`. The S3 PUT starts the pipeline. The folder stays the place a person browses by hand; S3 is where the system does its work. You don't lose the original — it stays in Drive too.

This lane covers the cases where you already have a recording on your laptop — a phone call you recorded, a webinar export, a voice memo — and you just want it in the archive. Drag, drop, and the next search will find it.

Lane 2: forwarding (the lane most teams actually use)

Set up a dedicated inbound address — something like `archive@your-company.com` — via Amazon SES. When a meeting tool emails you "your recording is ready" with the file attached, or when a teammate has a recording they want kept, they forward it to that address and the archive takes it from there. SES writes the raw email to `s3://tx-raw-mime/`. The S3 PUT triggers a parser Lambda that walks the email, finds the audio or video attachment (or a link to it), pulls the file, and stores it in `s3://tx-audio/` — the same place Lane 1 lands. From there it's the same pipeline.

If the forward carries a link instead of an attachment (some tools email a download link rather than the file), the parser follows the link, downloads the recording, and stores it. The original email is kept in S3 for audit, so you can always see how a given recording got in.

Lane 3: meeting-tool pull

Most recordings these days are born in a meeting tool — the video-call app the team already uses. Those tools keep finished recordings in their own cloud and offer an API to list and download them. Lane 3 is a small `connector` Lambda that runs on a schedule (every couple of hours), asks the meeting tool for recordings finished since the last run, downloads any new ones to `s3://tx-audio/`, and lets the pipeline take over. The meeting tool's credentials live in Secrets Manager.

This is the lane that makes the archive feel automatic. Nobody has to remember to upload anything — a recorded call simply shows up in the archive a couple of hours later, transcribed and searchable. A team that doesn't use a supported meeting tool loses nothing; Lanes 1 and 2 still cover them.

Transcribe, then file

Whatever lane the file came through, the rest is the same. The S3 PUT on `tx-audio` starts an Amazon Transcribe job. Transcribe turns the speech into text, labels who spoke each part (speaker one, speaker two, and so on), and stamps a time on every word. When the job finishes, a filing Lambda reads the result and does three plain-Python things: it takes the recording date from the file's metadata or its name; it matches the speaker labels and any "invited" list against the people aliases in the rules doc, so the catalogue says "Maria and the Acme

buyer” rather than “speaker one and speaker two”; and it tags a topic with a short keyword pass against the tag list. It writes one row to the `tx-catalogue` DynamoDB table — title, date, people, topic, access tag, transcript link — and saves the full transcript to `s3://tx-transcripts/`.

No model runs in this step. Filing is deterministic on purpose: the same recording always files the same way, and a person can correct a mis-tagged topic or a mismatched speaker by editing the rules doc, with no deploy. The next post is where the AI starts to earn its place — turning that transcript into something you can search by meaning.

Why everything funnels to one pipeline

Three doors in, but only one pipeline behind them. That’s deliberate. If each lane filed recordings its own way, “why is this recording tagged wrong?” would mean checking three code paths. Funneling every file through the same S3 bucket means transcription, filing, and indexing happen exactly once, the same way, no matter how the recording arrived. The lanes are first-class for getting recordings in; they all hand off to the same pipeline the moment the file lands.

Next post: how the filed transcript becomes searchable — split into timed chunks, turned into vectors, and written to S3 Vectors so a question can match by meaning.

PART 3 OF 7

JUNE 7, 2026 PART 3 OF 7 · TRANSCRIPTION ARCHIVE SERIES ~5 MIN READ

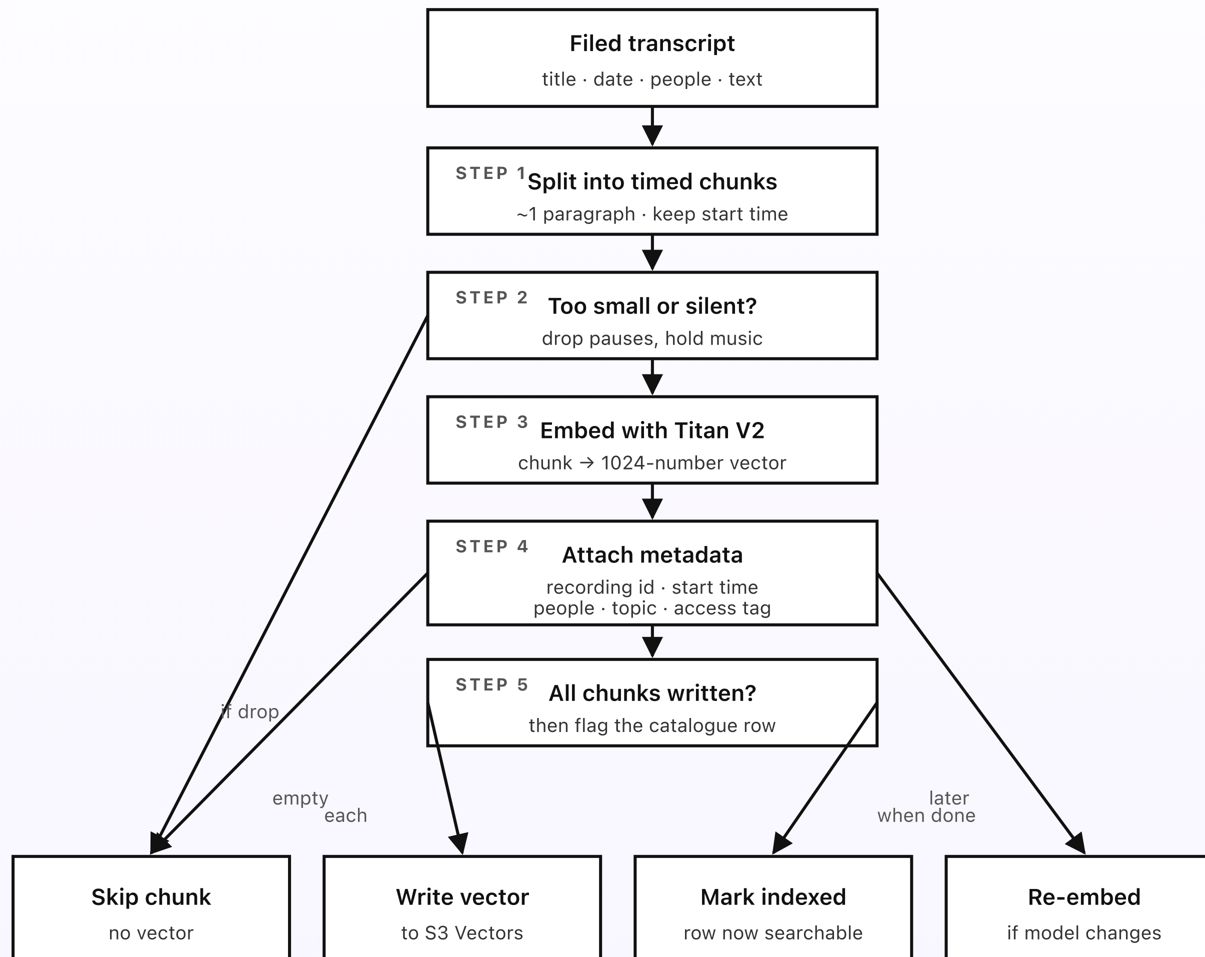
How a transcript becomes searchable

A transcript is a wall of text. Searching it for exact words misses the moment somebody said the same thing a different way — “we need it before year-end” won’t match a search for “deadline.” To fix that, the archive turns each transcript into something that can be matched by meaning. It splits the transcript into short timed chunks, turns each chunk into a vector, and writes the vectors to S3 Vectors. The chunking is plain Python. The embeddings are the only model call. After this step, the recording is searchable.

KEY TAKEAWAYS

- A vector is a list of numbers that captures meaning; two ways of saying the same thing land near each other.
- Each transcript is split into short chunks, every chunk carrying its start time in the recording.
- Titan Text Embeddings V2 turns each chunk into a 1024-number vector — the one model call in this step.
- Vectors land in S3 Vectors, each tagged with its recording, its timestamp, and its access tag.
- Indexing runs once per recording. Search later is cheap because the hard work is already done.

The indexing flow, per recording



Indexing runs once per recording — search later is cheap because the hard work is done.

Fig 3. The indexing flow, per recording. Split into timed chunks, drop the empty ones, embed each with Titan V2, attach metadata, and write to S3 Vectors. Once every chunk is written, the catalogue row is flagged searchable.

Chunking: short, timed, and a little overlapping

A whole transcript is too big to embed as one vector — the meaning of an hour-long call doesn’t fit in a single list of numbers. So the transcript is cut into short passages, roughly a paragraph each, on natural speaker turns and sentence boundaries. Every chunk keeps the start time of its first word, because that timestamp is what lets search later jump straight to the moment in the audio.

Chunks overlap a little — each one carries the last sentence or two of the chunk before it. That overlap means a thought that spans a chunk boundary (“...and the budget, which by the way, is firm at fifty thousand”) is still captured whole in at least one chunk, instead of being split in a way that hides it from search. The chunk size and overlap are plain settings in the rules doc; the defaults work for normal meetings.

Very short chunks and silent stretches — long pauses, hold music, the thirty seconds before everyone joined — are dropped before embedding. There’s no point spending a model call to index “[silence].”

Embedding: turning words into numbers that mean something

Each kept chunk is sent to Titan Text Embeddings V2, which returns a vector — a list of 1024 numbers. The trick of embeddings is that the numbers capture *meaning*: two chunks that say the same thing in different words land close together in that 1024-dimensional space, and two chunks about different things land far apart. That's what lets a search for "deadline" find a chunk that only ever said "before year-end." The same model will turn the search question into a vector later, so questions and chunks are measured on the same ruler.

Embedding is the one model call in this step, and it's a cheap one — a fraction of a cent per chunk. A typical hour-long meeting is a few dozen chunks, so indexing a recording costs a cent or two. And it happens exactly once. After that, the recording can be searched as many times as you like for almost nothing.

Writing to S3 Vectors

The vectors go into S3 Vectors, AWS's built-in vector store. Each vector is written with metadata attached: the recording id, the chunk's start time, the people, the topic, and — importantly — the access tag. That access tag is what lets search filter results by who's allowed to see them before any answer is written, which Part 5 covers in detail. Storing the access tag *on the vector* means the filter happens at the search itself, not as an afterthought.

S3 Vectors is the right fit here because the archive's search volume is bursty and low — a handful of searches a day, not thousands a second. You pay for the storage and the searches you actually run, with no always-on index server humming in the background. For an SMB archive that mostly sits quiet and occasionally gets a question, that economics is exactly right.

Re-indexing without re-transcribing

The full transcript is kept in S3 even after indexing. That matters for one reason: if the embedding model is upgraded down the line, or you change the chunk size, you can re-index the whole archive straight from the stored transcripts — no need to re-transcribe, which is the expensive part. Transcription happens once per recording, ever; indexing can be redone cheaply whenever it's worth it.

Next post: how a plain-language question turns into a vector, matches the closest chunks, gets filtered by access, and comes back as a short answer with a direct quote linked to the exact second.

PART 4 OF 7

JUNE 7, 2026 PART 4 OF 7 · TRANSCRIPTION ARCHIVE SERIES ~5 MIN READ

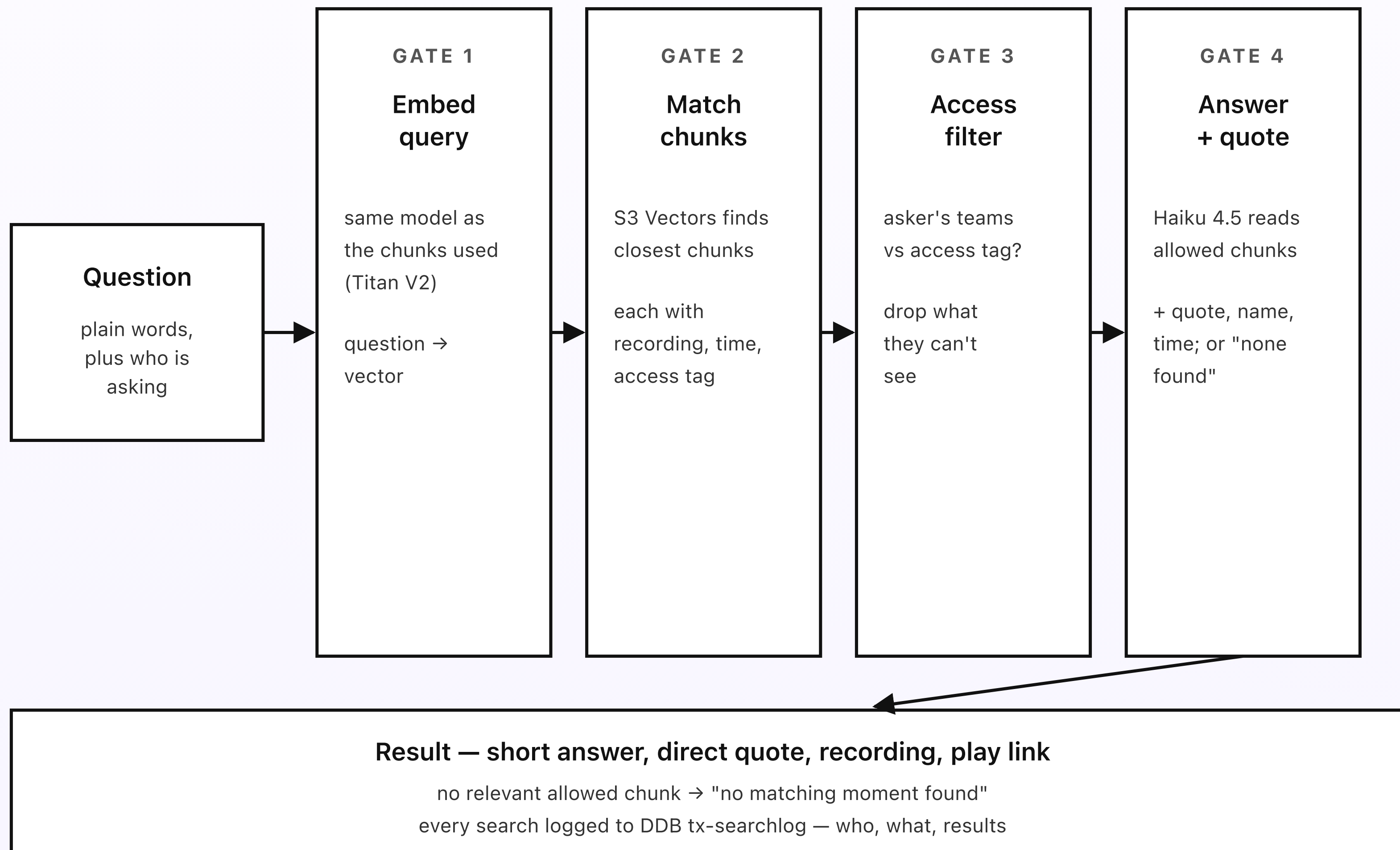
How a question finds the moment

Someone types “what did the Acme buyer say about the deadline?” into the search box. The archive has to turn that into a vector, find the closest chunks across the whole back-catalogue, drop anything the asker isn’t allowed to see, and write a short answer with a direct quote that links to the exact second. Get any of those wrong and the search is worse than useless: a quote from a call they can’t open, an answer with no source, a wall of half-matches. Four small gates sit between the question and the answer.

KEY TAKEAWAYS

- The question is embedded with the same Titan V2 model the chunks used, so they're measured the same way.
- S3 Vectors returns the closest chunks, each with its recording, timestamp, and access tag.
- The access filter drops anything the asker can't see before the answer is ever written.
- Haiku 4.5 writes a short answer with a direct quote — grounded only in the chunks it was handed.
- The result links to the exact second in the audio, and the whole search is logged.

Four gates on every search



Access is checked before the answer — a restricted quote never reaches the wrong person.

Fig 4. Four gates between the question and the answer. Embed the query. Match the closest chunks. Filter by access. Compose with a quote. Then ship the answer with a play link and log the search.

Gate 1: embed the query

The question is sent to the same Titan Text Embeddings V2 model that turned every chunk into a vector back in Part 3. That sameness is the whole point: because the question and the chunks are measured by the same model, “closeness” in that 1024-number space actually means “about the same thing.” A question embedded by a different model couldn’t be compared to the chunks at all — it would be measuring with a different ruler.

This is a single, cheap model call — a fraction of a cent. It’s the only embedding cost on the search path, because the chunks were already embedded once, at index time.

Gate 2: match the closest chunks

The query vector goes to S3 Vectors, which returns the handful of chunks closest to it across the whole archive — usually the top ten or twenty. Each returned chunk carries its metadata from index time: which recording it came from, its start time in that recording, the people, the topic, and its access tag. At this point the system has candidates, but it hasn’t yet checked whether the asker is allowed to see them, and it hasn’t written a word of answer. It just has a short list of the most relevant moments in the back-catalogue.

Matching by meaning is what makes this better than keyword search. The question “did they push back on price?” finds the moment somebody said “that number feels high for what we’re getting” — a match no keyword search for “price” would ever make.

Gate 3: the access filter

Before anything is shown or summarized, the search drops every candidate chunk the asker isn’t allowed to see. The request carries who is asking and which teams they belong to; each chunk carries an access tag. If the tags don’t match — a salesperson asking about an HR recording, say — the chunk is removed from the list. This happens *before* the answer model ever sees the chunks, which is the only safe order: you can’t accidentally quote from a recording that was filtered out before the quote was written. Part 5 covers the access model in full; the important thing here is *where* the filter sits in the flow.

If the filter removes everything — the only relevant moments are in recordings the asker can’t see — the search returns “no matching moment found” and logs the empty result. It never hints at what it hid.

Gate 4: compose the answer, then ship

The top few surviving chunks are handed to Claude Haiku 4.5 with a short, strict instruction: “Answer the question using only these chunks. Quote directly. Name the recording and the time. If the chunks don’t answer the question, say so — do not guess.” Grounding the model in only the retrieved chunks is what keeps the

answer honest: it can't invent a quote that isn't in the transcript, because the only material it's given is the transcript.

The result that lands in the search box is a short answer ("On the April 2 Acme call, their VP said they needed to go live before June 30"), a direct quote, the recording's name and date, and a play link that jumps to the exact second — built from the chunk's start time. The asker hears it in the speaker's own voice if they want to.

Every search — question, asker, the recordings that came back, and the timestamp — writes a row to `tx-searchlog` in DynamoDB. That log is both an audit trail and a quiet signal: if the same question keeps returning nothing, maybe a recording never got filed, or a topic tag is wrong.

Why the gates exist

None of these gates are exotic. They're the order a careful person would follow if they were doing the search by hand: figure out what's being asked, find the relevant moments, set aside anything you're not allowed to share, and then answer with the actual words and where to find them. Putting them in code as four small steps — with access checked *before* the answer is written — makes the safe behavior part of the design, not something you're trusting each search to remember.

Next post: the access model in full — how recordings get tagged, how sensitive ones stay out of open search entirely, and how the search log gives you a years-long record of who asked what.

PART 5 OF 7

JUNE 7, 2026 PART 5 OF 7 · TRANSCRIPTION ARCHIVE SERIES ~5 MIN READ

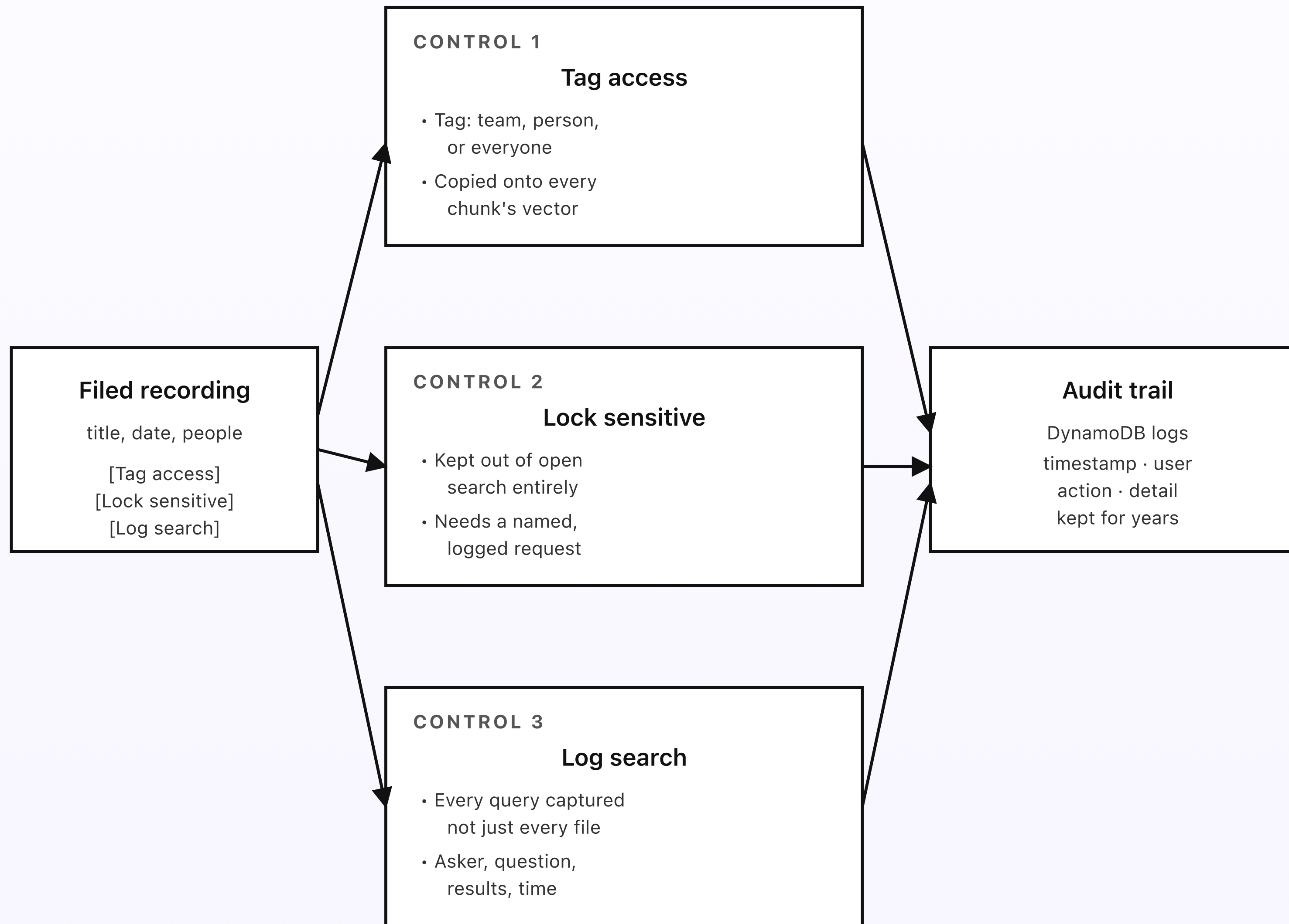
How the archive stays private

An archive of everything your business ever said on a call is powerful — and dangerous if anyone can search all of it. The sales team shouldn't surface an HR call. A contractor shouldn't find a board discussion. So the archive is built around three controls: every recording carries an access tag, search filters by that tag before any answer is written, and sensitive recordings can be kept out of open search entirely. On top of that, every search is logged. This post walks through all three, and what the log gives you.

KEY TAKEAWAYS

- Three controls per recording: *tag access* (who may see it), *lock sensitive* (keep it out of open search), *log search* (record every query).
- The access tag is stored on every chunk's vector, so the filter happens at the search itself.
- Locked recordings never appear in open search; they need a named, logged, direct request.
- Every search writes a row to the audit trail — who asked, what, what came back, and when.
- The default is private: a recording with no tag is visible only to its owner until someone sets one.

Three controls on every recording



The default is private — an untagged recording is visible only to its owner.

Fig 5. Three controls per recording, all feeding one audit trail. Tag access decides who may see it. Lock sensitive keeps the most private recordings out of open search. Log search captures every query. Every decision is recorded.

Control 1: tag access (who may see it)

Every recording gets an access tag when it's filed. The tag is one of three kinds: a *team* ("sales," "support," "leadership"), a *named person*, or *everyone*. The rules doc sets defaults — recordings from the sales meeting tool default to the sales team, recordings forwarded by a manager default to that manager — and a person can override the tag on any recording by editing the catalogue.

The important detail is *where* the tag lives. When the transcript is indexed (Part 3), the access tag is copied onto every chunk's vector. So when search runs (Part 4), the filter can drop disallowed chunks at the match step itself, without a second lookup. The asker's teams are checked against each chunk's tag, and anything they can't see is gone before the answer model is even called. A salesperson searching the whole archive simply never sees a sentence from an HR call — not because the answer hid it, but because those chunks were filtered out before any answer existed.

A recording with no tag at all defaults to its owner only. The system fails closed: when in doubt, fewer people can see it, not more.

Control 2: lock sensitive (out of open search)

Some recordings shouldn't turn up in a broad search even for people who could technically be allowed — a termination call, a legal discussion, a board session. For those, the tag isn't enough; you don't want them surfacing from a vague question that happened to match a phrase. So a recording can be marked *sensitive*. Sensitive recordings are still transcribed and indexed, but their chunks are flagged so the normal search box never returns them, no matter who's asking or how close the match is.

Reaching a locked recording takes a deliberate, named request — an authorized person opening that specific recording by id, which is itself logged. The point is that the most private material can't leak out of an innocent broad search. You have to mean to open it, and the system remembers that you did.

Control 3: log search (who asked what)

The first two controls decide what comes back. The third records what was asked. Every search — the asker, the exact question, the recordings that were returned, and the timestamp — writes a row to the `tx-searchlog` table. This is per *query*, not per recording: even a search that returned nothing is logged, because "who went looking for the layoff discussion" is itself worth knowing.

The log earns its place in three ways. It's an audit trail — if a quote turns up somewhere it shouldn't, you can see who searched for it and when. It's an early-warning system — a spike in searches for a sensitive topic is a signal worth a human glance. And it's a quality signal — if the same reasonable question keeps returning nothing, a recording probably never got filed or a topic tag is wrong. The

log is written by a path with append-only permission, so a search can record itself but can’t rewrite history.

How the three stack up

The controls are layers, not alternatives. Tag access decides the normal who-sees-what. Lock sensitive removes the riskiest recordings from open search no matter the tag. Log search records every query against both. A recording can be tagged to the sales team, and that’s the everyday control. A recording can additionally be locked, and now even sales can’t stumble on it. And every attempt to find either is written down. Each layer is simple on its own; together they let an SMB hold sensitive recordings in the same archive as everyday ones without turning the search box into a leak.

None of this replaces good judgment about what to record in the first place. But it means the archive earns trust the way a careful filing clerk would: it knows who’s allowed in which drawer, it keeps the locked drawer locked, and it writes down every time someone comes asking.

Next post: the cost breakdown. The whole archive runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why transcription dominates the bill.

PART 6 OF 7

JUNE 7, 2026 PART 6 OF 7 · TRANSCRIPTION ARCHIVE SERIES ~3 MIN READ

What the transcription archive costs

The archive does its expensive work once per recording, not once per search. Each recording is transcribed and indexed when it arrives; after that, searching it is almost free. So the bill tracks how many recordings come in, not how often people ask questions. Transcription is the biggest line by a wide margin. Embeddings, the answer model, and the vector store are small slivers. At typical SMB volume the whole thing is a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $4/month at typical SMB volume (about 60 recordings, ~40 hours of audio).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- Transcription dominates the bill — it runs once per recording and scales with hours of audio.
- Embeddings, the answer model, and S3 Vectors are small; searching is cheap because indexing already happened.
- At 250 recordings a month the bill is around $14. At 350 it's around $19.

Cost at three volumes

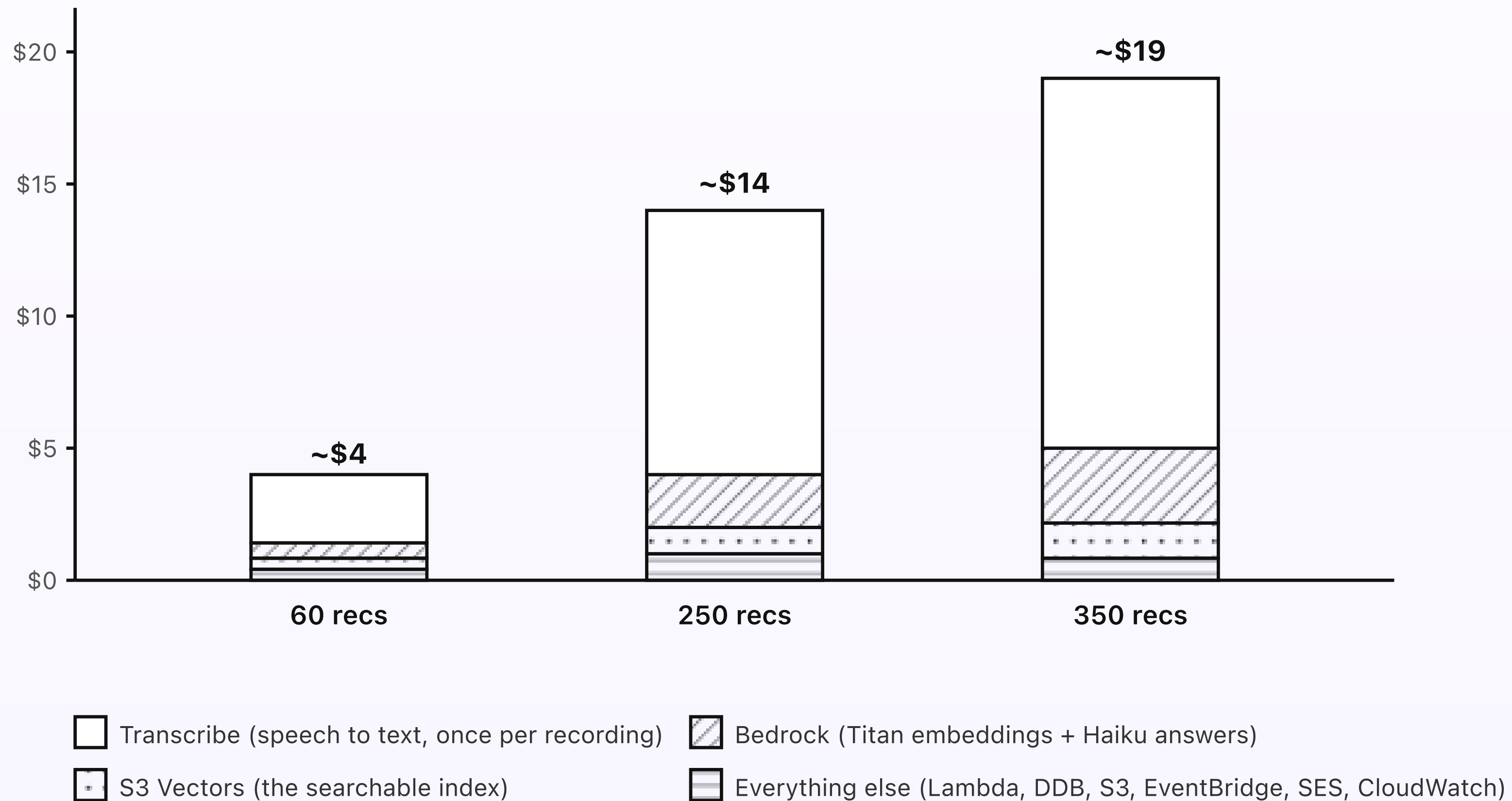


Fig 6. Monthly cost at three recording volumes. Transcribe dominates because it runs once per recording and tracks hours of audio. Bedrock, S3 Vectors, and the everything-else bucket stay small — embeddings run once and the answer model fires only on real searches.

Where the dollars actually go

Amazon Transcribe (the bulk). This is the big line, and it's priced per minute of audio. A typical SMB recording a couple of dozen hours of calls and meetings a month pays a few dollars for it. Transcribe runs exactly once per recording — never again, no matter how many times the recording is searched — so the cost tracks how much audio comes in, not how busy the search box is. The standard tier is fine for most calls; the cheaper batch tier suits the meeting-tool pull lane where there's no rush.

Bedrock (embeddings + answers). Two callsites. Titan Text Embeddings V2 runs once per chunk at index time — a few dozen chunks per recording, a fraction of a cent each, so a cent or two per recording. Claude Haiku 4.5 fires only when somebody actually searches: a few thousand input tokens (the retrieved chunks) and a couple hundred output tokens (the short answer), so a fraction of a cent per search. Even a busy team searching dozens of times a day keeps the answer-model cost under a dollar a month.

S3 Vectors. The searchable index. You pay for the vectors you store and the searches you run, with no always-on index server. A few thousand vectors at SMB scale is cents a month; the per-search cost is tiny. This is the part that would be expensive with an always-running search cluster — and isn't, because it's pay-per-use.

Lambda runtime. Every step is a short Lambda: the sync lanes, the filing step, the indexer, the search handler. None run long, none run often. Under a dollar a month at all three volumes.

DynamoDB on-demand. Three small tables: `tx-catalogue`, `tx-searchlog`, and the access records. A read or two per search, a write per recording and per query. Pennies a month at any of these volumes.

S3 storage + SES. The audio, the transcripts, and the raw forwarded emails. Audio is the heaviest, but a month of SMB recordings is a few gigabytes — cents — and old audio can move to a cheaper storage class. SES inbound for the forwarding lane is $0.10 per thousand messages: negligible.

What doesn't cost money

- **API Gateway.** Replaced by a Lambda Function URL for the search box.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate, no running search cluster. Everything is per-use.
- **Re-transcription.** Transcripts are kept, so re-indexing on a model upgrade never re-runs the expensive speech-to-text step.
- **Models on every search.** The embeddings are done at index time; a search is one cheap query embedding plus one short Haiku call.

How the cost scales

Transcribe and embeddings grow with how much audio you bring in, because each recording is processed once. The answer model grows with how often people search — a separate, smaller dial. Storage grows slowly with the back-catalogue. So the bill at 1,000 recordings a month is around $50, dominated by

transcription; at 2,500 it’s around $120. Past those volumes you’d look at the batch transcription tier and lifecycle rules to move old audio to cold storage, but those are tunings, not redesigns.

Set an AWS Budgets alarm at $25/month so anything unusual pages you before the bill matters. The archive’s normal-volume bill stays well under that ceiling.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, the Transcribe job config, Lambda inventory, IAM scopes, the S3 Vectors index, Bedrock model IDs, and the DynamoDB schemas.

Build it for real → shop.allanninal.dev/w/transcription-archive

PART 7 OF 7

JUNE 7, 2026 PART 7 OF 7 · TRANSCRIPTION ARCHIVE SERIES ~8 MIN READ

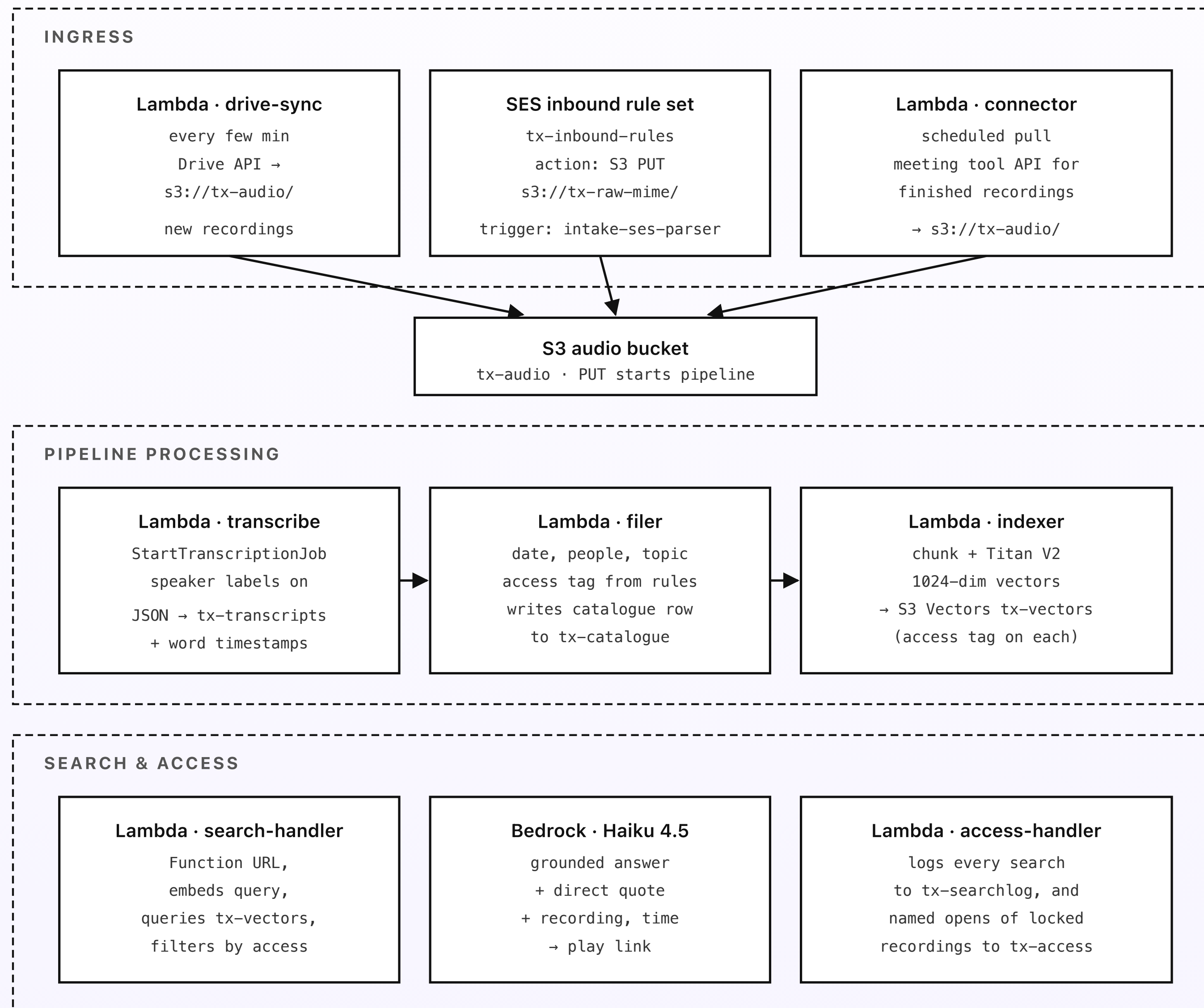
Engineering reference: the transcription archive architecture

Same system, drawn for engineers. Region, service names, resource identifiers, the Amazon Transcribe job config, Bedrock model IDs, the S3 Vectors index layout, Lambda inventory, IAM scopes, the SES inbound rule set, and the DynamoDB schemas. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). Amazon Transcribe, S3 Vectors, Bedrock cross-Region inference, and SES inbound are all available there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for an SMB is a search that returns nothing for an hour, not a regional outage. One AWS account dedicated to the archive (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system. Data residency matters here: recordings can be sensitive, so pick the region that satisfies your contracts and keep audio, transcripts, and vectors all in it.

Topology



Every answer is grounded in retrieved chunks — and every search is logged.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the audio bucket), pipeline processing (transcribe, file, index), search and access (the query resolves to a grounded answer and the search is logged). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

All Lambdas use the `arm64` architecture, the smallest memory size that meets latency targets (typically 256 MB), Python 3.14 runtime, and CloudWatch Logs at 7-day retention. Each function has its own least-privilege IAM role. None run inside a VPC.

- **`drive-sync`** — EventBridge Scheduler target, fires every few minutes. Uses the Google Drive API (service-account credentials in Secrets Manager under `tx/drive/sa`) to list the watched folder and copy new audio/video objects to `s3://tx-audio/`, recording a synced-files marker so it never re-copies. The same pattern syncs the rules and access docs to `s3://tx-rules-source/`. Memory: 256 MB. Timeout: 60 s.
- **`intake-ses-parser`** — S3 PUT trigger on `s3://tx-raw-mime/`. Parses MIME, locates the audio/video attachment (or a download link), pulls the media, and stores it in `s3://tx-audio/`. Large attachments are streamed, not buffered. Keeps the raw MIME for audit. Memory: 512 MB. Timeout: 120 s.
- **`connector`** — EventBridge Scheduler target, every two hours. Calls the meeting tool's cloud API (OAuth token in Secrets Manager under `tx/meeting/oauth`) for recordings completed since the last cursor, downloads new ones to `s3://tx-audio/`, and advances the cursor in Parameter Store.

Handles the tool's pagination and rate limits; backs off on 429. Memory: 512 MB. Timeout: 300 s.

- **`transcribe`** — S3 PUT trigger on `s3://tx-audio/`. Calls `StartTranscriptionJob` with `ShowSpeakerLabels=true`, automatic language identification (or a fixed language from config), and output to `s3://tx-transcripts/<recording-id>.json`. Uses the batch tier for connector-sourced jobs (no latency pressure) and the standard tier for forwarded ones. Memory: 256 MB. Timeout: 30 s (the job itself runs async in Transcribe). *No Bedrock calls.*
- **`filer`** — triggered by the Transcribe job-completion event on EventBridge. Reads the transcript JSON, derives the recording date from object metadata, maps speaker labels and invitee lists to people aliases from the rules doc, tags a topic via a keyword pass, and resolves the access tag from the rules defaults. Writes one row to `tx-catalogue` and emits `tx.filed`. Memory: 256 MB. Timeout: 60 s. *No Bedrock calls.*
- **`indexer`** — EventBridge rule on `tx.filed`. Chunks the transcript (~1 paragraph, sentence-aligned, small overlap, each chunk carrying its first-word start time), drops empty/silent chunks, calls Titan Text Embeddings V2 (`amazon.titan-embed-text-v2:0`) per chunk for a 1024-dim vector, and writes vectors with metadata (`recording_id`, `start_ms`, `people`, `topic`, `access_tag`, `sensitive`) to the S3 Vectors index `tx-vectors`. Flags the catalogue row searchable when all chunks land. Memory: 512 MB. Timeout: 120 s.
- **`search-handler`** — Lambda Function URL, `AuthType: AWS_IAM` fronted by your identity provider, or a signed session for the internal search UI. Embeds

the query with Titan V2, queries `tx-vectors` (top-k with a metadata filter on `sensitive=false`), drops chunks whose `access_tag` the caller's teams don't cover, then calls Claude Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) with the surviving chunks and a strict grounding prompt. Returns answer, quote, recording, and a deep link built from `start_ms`. Writes a `tx-searchlog` row. Memory: 512 MB. Timeout: 30 s.

- **`access-handler`** — Lambda Function URL for named opens of locked (sensitive) recordings. Verifies the caller is authorized for that specific `recording_id`, returns the transcript or a signed audio URL, and writes a `tx-access` row. This is the only path that can surface a sensitive recording, and it is always logged. Memory: 256 MB. Timeout: 15 s.
- **`digest`** — EventBridge Scheduler target, weekly Monday 9am. Reads `tx-searchlog` and `tx-catalogue` for the week; emails an admin summary via SES (new recordings filed, top searches, empty-result questions worth investigating). No Bedrock; plain summary table. Memory: 256 MB.

Storage

- **S3 · `tx-audio`** — source recordings. Versioning enabled. Lifecycle to Glacier Instant Retrieval at 60 days; no auto-expiry by default (recordings are the record). SSE-KMS with a dedicated key.
- **S3 · `tx-transcripts`** — Transcribe JSON output, kept so the archive can be re-indexed without re-transcribing. Versioning enabled. SSE-KMS.
- **S3 · `tx-raw-mime`** — raw inbound MIME from forwarded recordings, for provenance. Lifecycle to Glacier at 30 days; expiry at 7 years.

- **S3** · `tx-rules-source` — mirrored rules and access docs as plain text. Versioning enabled.
- **S3 Vectors** · `tx-vectors` — the searchable index. 1024-dim vectors from Titan V2, one per kept chunk. Metadata per vector: `recording_id`, `start_ms`, `people`, `topic`, `access_tag`, `sensitive`. Queried top-k with a metadata pre-filter.
- **DynamoDB** · `tx-catalogue` — one row per recording. PK `recording_id`; attributes: `title`, `date`, `people`, `topic`, `access_tag`, `sensitive`, `transcript_key`, `audio_key`, `indexed`. On-demand. GSI on `date` for browse.
- **DynamoDB** · `tx-searchlog` — one row per query. PK `(user_id, ts)`; attributes: `query`, `returned_ids`, `result_count`, `latency_ms`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB** · `tx-access` — one row per named open of a locked recording. PK `(recording_id, ts)`; attributes: `user_id`, `reason`, `granted_by`. On-demand. No TTL.

Amazon Transcribe

- **Job config.** `StartTranscriptionJob` with `ShowSpeakerLabels=true` and `MaxSpeakerLabels` tuned to room size; `IdentifyLanguage=true` unless a fixed language is set in config. Output to `s3://tx-transcripts/`. Custom vocabulary (product names, people, acronyms) raises accuracy on domain terms.

- **Tiering.** Connector-sourced jobs use the batch path (no latency pressure); forwarded recordings use standard. PII redaction can be enabled per access tag so transcripts of sensitive recordings store redacted text by default.
- **Completion.** Transcribe emits a job-state-change event to EventBridge; the `filer` Lambda triggers on `COMPLETED` and on `FAILED` writes the recording to a dead-letter prefix and alerts.

Bedrock

- **Embeddings.** `amazon.titan-embed-text-v2:0`, 1024-dim, normalized. Two callsites: the `indexer` (one call per chunk at index time) and the `search-handler` (one call per query). The query and the chunks must use the same model and dimension.
- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. One callsite: the `search-handler`, composing the grounded answer. Sonnet 4.6 is not used — the answer is a short, grounded summary of a few chunks, well within Haiku's range, and the cost difference matters at search volume.
- **Quotas.** Default account quotas are more than enough at SMB volume. The expensive work is Transcribe, not Bedrock.

EventBridge and Scheduler config

- **`tx-drive-sync`** — `rate(5 minutes)`. Target: `drive-sync` Lambda.
- **`tx-connector`** — `rate(2 hours)`. Target: `connector` Lambda.

- `tx-weekly-digest` — `cron(0 9 ? * MON *)` in TZ. Target: `digest` Lambda.
- **Transcribe completion rule** — EventBridge rule on `aws.transcribe` Job State Change → target `filer` Lambda.
- **`tx.filed` rule** — custom-bus rule on the `filer`'s emitted event → target `indexer` Lambda.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `archive.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `tx-inbound-rules`: one rule with recipient `archive@your-company.com` → spam scan → S3 PUT to `s3://tx-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for the weekly digest: verify a sender identity at `archive-bot@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request.

IAM (least privilege per Lambda)

Each Lambda has its own role with policies scoped to exact ARNs. Sketch:

- **transcribe role:** `s3:GetObject` on `tx-audio`; `transcribe:StartTranscriptionJob`; `s3:PutObject` on `tx-transcripts`; `kms:Decrypt` + `GenerateDataKey` on the archive key. *No* `bedrock:*`.
- **filer role:** `s3:GetObject` on `tx-transcripts` and `tx-rules-source`; `dynamodb:PutItem` on `tx-catalogue`; `events:PutEvents` on the custom bus.

No `bedrock:*`.

- **indexer role:** `s3:GetObject` on `tx-transcripts`; `bedrock:InvokeModel` on the Titan ARN; `s3vectors:PutVectors` on `tx-vectors`; `dynamodb:UpdateItem` on `tx-catalogue` (the indexed flag).
- **search-handler role:** `bedrock:InvokeModel` on the Titan ARN and the Haiku ARN; `s3vectors:QueryVectors` on `tx-vectors`; `dynamodb:PutItem` on `tx-searchlog`; `dynamodb:GetItem` on `tx-catalogue`. No write access to audio, transcripts, or vectors.
- **access-handler role:** `dynamodb:PutItem` on `tx-access`; `dynamodb:GetItem` on `tx-catalogue`; `s3:GetObject` + presign on `tx-audio` and `tx-transcripts` scoped per-request to the opened `recording_id`.
- **drive-sync / connector / intake-ses-parser roles:** `secretsmanager:GetSecretValue` on the relevant secret; `s3:PutObject` on `tx-audio` (and `tx-rules-source` for drive-sync); outbound network to the Google or meeting-tool API only.

Search surface

The search box is a small static page that posts the query and the caller's identity token to the `search-handler` Function URL. Identity comes from your existing SSO (the Function URL is fronted by IAM auth or a short-lived signed session); the archive doesn't run its own user store. The response is rendered as a short answer, the quote in a blockquote, the recording title and date, and a play button whose link carries the `start_ms` so the audio element seeks straight to the moment. Locked recordings never appear here; opening one is a separate, authorized action through `access-handler`.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch metric for alerting.
- **Alarms:** Transcribe job failures > 0 in a day; indexer failures > 0 (an un-indexed recording is invisible to search); search-handler p95 latency > 4s; access-handler authorization failures > 5/hour.
- **X-Ray:** on for the `search-handler` only (the user-facing path); off elsewhere to save cost.
- **AWS Budgets:** $25/month threshold, alarm at 80% and 100%, posts to SNS topic `tx-cost-alarm` subscribed to the on-call admin's email.

Config and secrets

Service-account credentials for the Drive API live in Secrets Manager under `tx/drive/sa`; the meeting-tool OAuth token under `tx/meeting/oauth`. The connector cursor, the chunk-size and overlap settings, the topic and people-alias tables, the access defaults, and the SES sender identity all live in Parameter Store under `/tx/config/` (the larger tables as JSON in `tx-rules-source`, mirrored from Drive). The KMS key id for the archive and the index region are also config. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role — no long-lived keys — running AWS SAM. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for `tx-audio`, `tx-transcripts`, and `tx-rules-source` so a bad sync or edit can be rolled back, and keep the KMS key and the S3 Vectors index in the same region as the audio for data-residency reasons. Total deployable surface: around nine Lambdas, three DDB tables, one S3 Vectors index, four S3 buckets, a handful of EventBridge rules, one SES rule set, and one Budgets alarm.

That's the full system. Six narrative posts and this engineering reference. If you want to talk about adapting it for your business, see Work with me.